Chapter 1

Mathematical Model Transformations

1.1 Introduction

For most computer controlled systems, especially dependable, real–time systems for critical applications, an effective design process requires an early conceptual and architectural validation prior to the implementation in order to avoid costly re–design cycles. In order to have a guaranteed design quality, all relevant system characteristics have to be checked during this *system verification* phase. These parameters identify critical bottlenecks to which the system is highly sensitive.

The increasing need for effective design has necessitated the development of standardized and well-specified design methods and languages, which allow system developers to work on a common platform of design tools. The *Unified Modeling Language (UML)* is a visual specification language for pure software systems, as well as for embedded real-time systems (systems reactively interacting with their environment). The UML represents a collection of best engineering practises that have proven successful in the modelling of large and complex systems. Recently, UML has been regarded as the standard object–oriented modelling language.

1.1.1 Formal Methods in System Design

Formal methods are mathematics-based techniques offering a rigorous and effective way to model, design and analyze computer systems. They have been a topic of research (in projects like IOSIP [8], SafeRail [5], SpeciMen [7] or HIDE [2]) for many years with valuable academic results. However, their industrial utilization is still limited to specialized development sites, despite their vital necessity originating in the complexity of IT products and the increasing requirements for dependability and Quality of Service (QoS).

The use of formal verification tools (like SPIN [11] or PVS [21]) in IT system design is hindered by a gap between practice–oriented CASE tools and sophisticated mathematical tools. On the one hand, system engineers usually show no proper mathematical skills required for applying formal verification techniques in the software design process. On the other hand, even if a formal analysis is carried out, the consistency of the manually created mathematical model and the original system is not assured. Moreover, the interpretation of analysis results, thus the re–projection of the mathematical analysis results to the designated system is problematical. From the engineering point of view, the notion of dependability is a composite one necessitating the analysis of multiple mathematical properties by using different verification tools.

The aim of our ongoing research is to provide a provenly correct and complete, automated transformation between UML–based system models and formal mathematical verification tools for an effective software design.[28]

1.1.2 Mathematical Model Transformation

The step generating the input language of a target mathematical tool from the UML model of the system is denoted as *mathematical model transformation.* The inverse direction of model transformation (referred as *back–annotation*) is of immense practical importance as well when some problems (e.g. a deadlock) are detected during the mathematical analysis. After an automated back–annotation these problems can be visualized in the the same UML system model allowing the designer to fix conceptual bugs within his well–known UML environment.

Several semi-formal transformation algorithms have already been designed and implemented for different purposes (e.g formal verification of functional properties [16] and quantitative analysis of dependability attributes [3, 4, 6]). Unfortunately, this conventional way of model transformation lacked a uniform and precise description of transformation algorithms resulting in hand–written and rather ad hoc implementations (inconvenient for implementing complex transformations). Moreover, any formal proof of correctness and completeness aiming to verify these transformation scripts is almost impossible, thus their uncertain quality remains a quality bottleneck of the entire transformation based verification approach.

Thus, a model transformation system (avoiding these drawbacks) must fulfil at least the following user requirements.

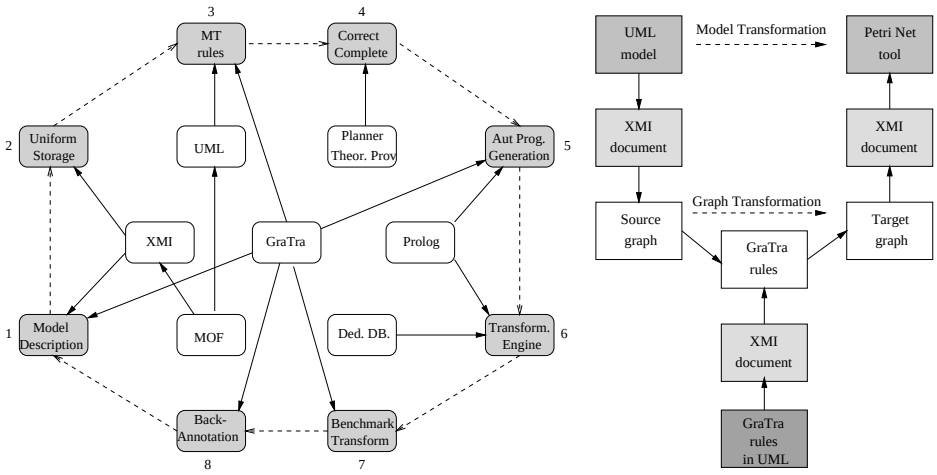
- A large number of model transformations are planned to be designed to perform dependability analysis in various application domains ranging from early evaluation methods based on Petri nets to model checking techniques using temporal logic as underlying mathematical model.
- "Mathematical" model transformations are not only designed by mathematicians but system engineers as well. Thus, these transformations must be defined by a visual, easy to understood formalism.
- The specification of a model transformation should be given in mathematically precise, unambiguous form.

1.1.3 VIATRA: A Visual Automated Model Transformation System

The process of model transformation is characterized by a model analysis round-trip illustrated in Fig. 1.1. Typically, a system designer and a transformation designer participates in such a round-trip with the following roles.

- A transformation designer specifies model transformations from UML to various mathematical models (like e.g. Petri nets, temporal logic). From his specification, a transformation algorithm is generated at compile time.
- A system analyst designs complex systems using UML as modelling language. During the software life cycles, he needs several verification steps to be performed running the previously generated model transformation programs.

Model description A well–defined transformation necessitates a uniform and precise description of source and target models. On the other hand, it should follow the main standards of the industry. For this reason, the **Meta Object Facility (MOF)** metamodelling techniques are used. MOF metamodels provide graphical means to define metaobjects for similarly behaving instances



(a) Architectural concepts (b) Technological summary

Figure 1.1: An overview of model transformation

in various domains by combining the expressive power of UML Class diagrams (concerning the structure) with the Object Constraint Language (OCL) for describing semantic issues. MOF metamodels are used as a basis for describing UML models (following the standard metamodel of UML) as well as mathematical structures (by creating non–standard metamodels for them).

A typical UML model contains more details than required for a specific mathematical analysis (for instance, documentation or use case diagrams are often of little importance). Thus, in the sequel, a UML model will only contain the relevant pieces of information with respect to a specific analysis, and this reduced model can be obtained from the original user–created system model by some filtering mechanism.

In VIATRA, filtering is expressed by metamodels. Exactly those constructs are regarded relevant (thus transformable) that are included in the metamodel of the source language (hence if specific constructs are irrelevant for one purpose, they are simply not included in the metamodel).

Uniform description of models The front–end and back–end of transformations (UML as the source model and a formal verification tool as the target model) is defined by a uniform, standardized description language of system modelling, that is, **XMI (XML Metadata Interchange)** [18] XMI is a special metamodel dependent collection of XML constructs providing an XML representation for arbitrary (MOF based) models.

XMI seems to be a natural choice as a large number of UML tool vendors provide a facility to export their models into XMI, moreover, several academic communities (e.g. Petri Net [12] the graph transformation community [25]) have started discussion to settle on a general XML based interchange format for their tools.

Designing model transformation rules The visual specification of model transformations is supported by **graph transformation** [22], which combines the advantages of graphs and rules into an efficient computational paradigm.

A **graph transformation rule** is a special pair of pattern graphs where the instance defined by the left hand side is substituted with the instance defined by the right hand side when applying such a rule (similarly to the well–known grammar rules of Chomsky in computational linguistics).

Model transformation rules (in the form of graph transformation rules) are specified by using the visual notation of UML. However, for obtaining a tool–independent transformation specification, the transformation rules will also be exported in an XMI based format, conforming to the approving standard of graph transformation systems [25].

Correctness and completeness of transformations Automated transformations necessitate an automated proof method aiming to verify that the generated target models are semantically correct. Moreover, each construct allowed in the source model should be handled by a corresponding rule. Instead of verifying the semantic correctness of individual target models (generated by some model transformation), our alternate solution puts the stress on the **correctness and completeness of transformation rules**, i.e. starting from a source model fulfilling some semantic criteria, the derivation steps should always keep these properties invariant for the target model.

Automated program generation Even if the description of the transformation is theoretically correct and complete, additionally, the source and target models are also mathematically precise, the implementation of these transformations has a high risk in the overall quality of a transformation system. As a possible solution, an **automatic generation of the transformation algorithm** is carried out including (automatically generated) programs for implementing visual transformation rules and control structures [9].

The transformation engine As being a logic programming language based on powerful unification methods, Prolog seems to be a suitable language for a prototype implementation of the transformation engine. Thus, the XMI based models and rule descriptions are translated into a Prolog graph notation serving as the input data and the program to be executed, respectively. After a successful prototyping phase, Prolog could be substituted with a more powerful but lower abstraction level language (like C++ or Java).

Benchmark transformations The model transformation system is planned to be used in real industrial applications. Several benchmark transformations have already been designed and implemented.

- Transforming the static aspects of UML models into timed Petri Nets for dependability analysis in an early phase of system design (discussed in [27]);
- Transforming UML Statecharts into Extended Hierarchical Automaton providing formal semantics for those diagrams (a formal description of the transformation [16]; presented in the current paper).
- Automatic program generation for visual control structures [9].

Back–annotation of analysis results The results of the mathematical model transformation are planned to be automatically back–annotated to the UML based system model. Thus,the system analysts are reported from conceptual bugs in their well–known UML notation. Unfortunately, the current version of UML does not directly support the representation of analysis traces. For instance, the sequence of fired statechart transitions that leads to a deadlock (according to the verification tool) completely lacks a fine–grained UML representation. Please note that as the results of an analysis may form a totally different model in contrast to their input specification (e.g. sequence of fired transitions instead of state machines), the problem of back–annotation might not be equivalent with an inverse transformation.

1.1.4 A Scenario of Model Transformation Engineering

The structure of the current paper is characterized by traversing the main steps in the entire process of model transformation design on a benchmark example. According to model transformation round–trip, such a process consists of the following steps.

1. **Creating metamodels**
   (a) As soon as the source (typically the Unified Modeling Language) and target (a formal verification method and tool) languages are determined the meta–models (a description defining the semi–formal semantics of a specific application domain) of both languages are constructed following the guidelines of the Meta Object Facility standard (Section 1.2).
   (b) Sample models conforming the metamodels may be constructed to verify and test the metamodelling step.
2. **Creating a source model**
   (a) A sample user UML model (a statechart describing dynamic behaviour in our benchmark) created by an arbitrary CASE tool (e.g. Rational Rose, Innovator, etc.) having an XMI export feature.
   (b) The well–formedness of the user model is checked.
   (c) The UML model is converted into the MOF Model based XMI model interchange format by using the standard UML DTD.
   (d) This XMI description is processed by a simple parser to build up a simple graph database.
3. **Designing the transformation**
   (a) Model transformation rules (in the form of graph transformation rules) are created and nested into transformation units (which provide means for modular construction).
   (b) These rules are specified visually (e.g. by overloading the syntax of UML).
   (c) The correctness and completeness of the transformation is proved.
   (d) The transformation code (which is Prolog during the prototyping phase and an object–oriented later on) is generated automatically.
   (e) The transformation is executed, the target description is generated.
4. **Formal mathematical analysis**
   (a) The mathematical analysis is performed (e.g. detecting dead–locks, verifying liveliness properties and specification consistency).
   (b) The analysis results are back–annotated to the UML system model by reference relations between the source and target objects.

1.1.5 Paper Organization

The rest of the paper is structured as follows.

- Section 1.2. summarizes the concepts of MOF metamodelling which will serve as a basis for describing models from arbitrary domain.

- In Section 1.3, the theoretical foundations of model transformation are introduced by creating graphs from MOF based models.
- Section 1.4. provides a brief informal introduction to a benchmark example, i.e. transforming UML statecharts into extended hierarchical automaton for obtaining an operational semantics for semi-formal statechart descriptions.
- Afterwards, in Section 1.5, this benchmark transformation is specified formally, by means of graph transformation rules and units.
- Section 1.6 is concerned with the implementation of such model transformations by describing a general and highly language independent method for automatic program generation.
- Finally, Section 1.7 concludes our paper.

1.2 MOF Metamodels

Nowadays, due to the large variety of systems, integration and standardization play a major role in the process of software design. Basic requirements concern the re–use of information models in various domains, which in turn necessitates a standardized description of these models in order to avoid incompatibility problems.

The standardization process of domain specific models typically consists of two steps. At first, the standardization committee should agree on a common high–level notation (including the identification of basic entities, their attributes and relations) by creating the metamodel — a model describing another model — of the specific domain. In order to keep the size of the metamodel manageable, tool (or approach) specific information is not included, thus, this first step determines "what to include" in the metamodel.

This metamodelling is supported by the Meta Object Facility standard of the Object Management Group (OMG) which provide a common basis for describing metamodels of arbitrary domain by combining the expressive power of UML Class diagrams (with respect to model structure) with the Object Constraint Language (OCL) for describing semantic issues.

At the second phase of standardization, the committee should decide "how to include" these concepts concentrating on such requirements as easy structure, extensionality and document validation. Moreover, since these models are typically used in a distributed and heterogeneous environment (i.e. the Internet), a standard model description format would be based on the novel standard of the web, the XML (eXtensible Markup Language).

Another OMG standard, the XMI (XML Metadata Interchange) was introduced to provide an automated mapping from MOF based metamodels to an extensible set of XML constructs. With this respect, the standardization committee may concentrate on the high–level metamodelling issues as its XML implementation may automatically be generated.

1.2.1 Concepts of Metamodelling

Thus, the concepts of metamodelling (illustrated in Figure 1.2) originates in the need for an effective design process of formal specification and modelling languages. The large number of similar languages — often supported nowadays by visual diagrams — necessitates a common model description language (called **meta–metamodel** or **MOF Model**).

The sentences of this top–level language (denoted as **metamodels**) are to describe the structure of domain specific information models. For instance, the metamodel of UML is to describe the

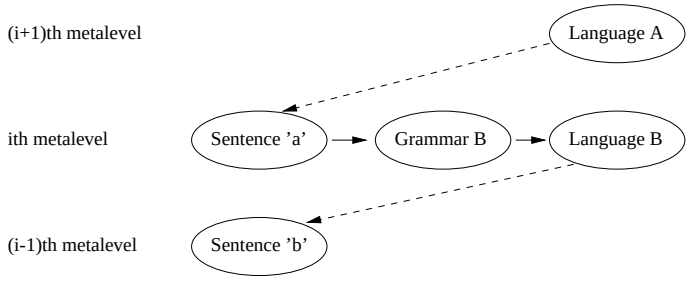


Figure 1.2: Meta–layers in language specification

major concepts of UML. However, these sentences of the common meta–metamodel may be regarded in turn as separate sub–languages (the language of UML, Petri Nets, etc.), thus they provide a grammar to describe sentences of a lower meta–level. These lower level sentences are denoted as **models** (e.g. the UML metamodel serves as a common grammar for describing different UML models as sentences).

As a result, a model hierarchy (summarized in Table 1.1) is available with at least four meta–layers.

Meta–level	MOF terms	Examples
M3	meta–metamodel	The MOF Model
M2	metamodel	UML Metamodel
M1	model	UML Models
M0	data	modelled systems

Table 1.1: MOF Metadata Architecture

Although, the metamodelling concepts are related mainly to UML and software models, the similar concepts can be applied for describing the structure of arbitrary mathematical models as such models use languages of lower abstraction level. A MOF metamodel of e.g. Petri Nets, or a finite automaton provides an easy–to–understand way to obtain a visual overview of the underlying mathematical structures. In our benchmark example (cf. Section 1.4), we try to demonstrate that even abstract mathematic structures can easily be understood by system engineers if they are given in a MOF notation.

1.2.2 The MOF Model

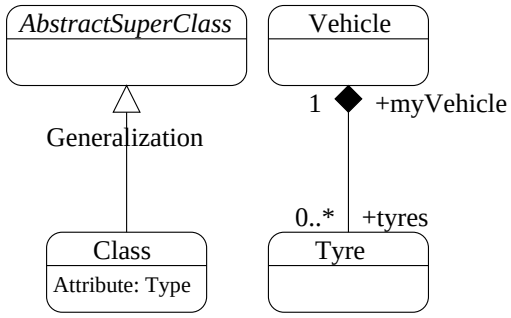
*The **MOF Model** is an abstract language for defining MOF metamodels* (such as the metamodel of UML itself). Although MOF and UML was designed for different purpose (i.e. metadata versus system modelling) the MOF Model and the core of the UML metamodel are closely related in their modelling concepts (classes for objects of similar structure; associations as relations between these classes; generalization, etc.). Therefore, the corresponding UML notation is being used as a notation for MOF–based metamodels. Nevertheless, in order to distinguish between the metamodel elements of UML and the basic constructs of the MOF Model, latter ones are printed in capital initials.

The main metamodelling constructs provided by the MOF (and used in the current paper) are the following (see also Figure 1.3 for the graphical notation).

- ***Classes*** *are used for identifying and describing M2 level meta–entities.* For instance, in case of traditional finite automaton, such meta–entities could be States, Transitions, Automaton, etc. Please note that instances of these meta–classes (concrete states) appear on M1 level thus it is a subject of a subsequent modelling phase. The structural features of Classes can be described by
  - **Attributes**: a value holder in an instance of the class;
  - **Generalization**: in this case the instance of a Class inherits its structure and behaviour from instances of other Classes.
  - **Abstract Classes**: these Classes may not have any instances (at a lower meta–level).
- ***Associations*** *are binary relations between Class instances.* Each Association has two **AssociationEnds** that may specify
  - **aggregation** semantics (when a Class instance is composed of several other Class instances)
  - **cardinality** (a Class instance may be in a specific relation with a limited number of instances)
  - **uniqueness** (the same instance must not appear twice in a model)
  - a **Reference** *that allows navigability of the Association's links* (i.e. Classes) from a Class instance when the Class is the type of an AssociationEnd
- The following constructs are also included in the MOF Model, however, they are not discussed in details in the current paper.
  - **Packages** are collections of related Classes and Associations and they support a modular composition and information hiding in a metamodel by nesting and importing other Packages.
  - **DataTypes** allow the use of basic and external types for Operation parameters and Attributes.
  - **Constraints** are used to associate semantic restrictions with other elements in a MOF metamodel by defining well–formedness rules for the metadata described by a metamodel. The **Object Constraint Language (OCL)** [19] is often used as a semi–formal language for expressing constraints, however, it still lacks a precise semantic basis.

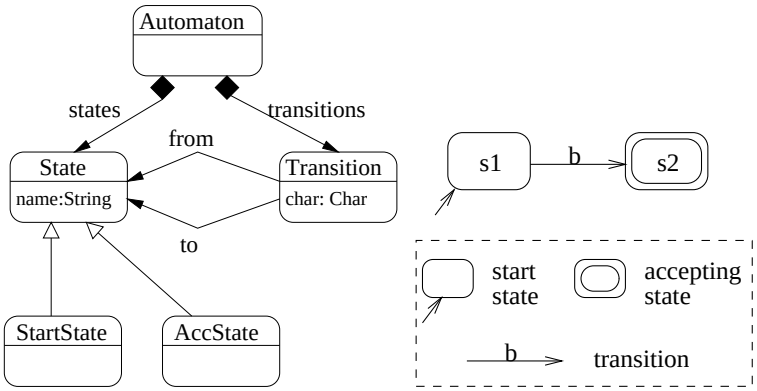
Example 1.2.1 In Figure 1.3, sample MOF metamodels illustrate the graphical notation of MOF.

- In Figure 1.3(a), an `AbstractSuperClass` (printed in italics) is introduced from which a sample `Class` is inherited. This `Class` has an `Attribute` of a specific `Type`.
- The metamodel in Figure 1.3(b) states that every instance of a `Vehicle` is composed of (note the black diamond) an arbitrary number of `Tyres`, while each instance of a `Tyre` may belong to a single `Vehicle` (instance) by the corresponding cardinalities (`0..*` and `1` respectively). The `Tyre` instances are accessible from the container `Vehicle` via the `tyres` AssociationEnd, while the Association is navigable via `myVehicle` in the opposite direction.



(a) Classes (b) Associations

Figure 1.3: Graphical MOF notation



(a) MOF metamodel (b) Sample automaton

Figure 1.4: Description of finite automaton

Example 1.2.2 A sample model and MOF metamodel of finite automaton are provided in Figure 1.4.

- According to the metamodel:
  - A finite automaton is composed of states and transitions (indicated by the corresponding MOF Classes `Automaton`, `State` and `Transition` and Aggregations `states` and `transitions`).
  - A `State` has a `name` as an Attribute, and may also be in turn either a start state (`StartState`) or an accepting state (`AccState`).
  - A transition has a character (`char`) of type `Char` as an attribute (taking its value from the alphabet). Moreover, a transition is leading from a state into another state (expressed by two associations `from` and `to`).
- In the sample automaton:
  - There is a start state called `s1` and an accepting state `s2`.
  - There is a transition from `s1` to `s2` triggered by character `b`.

Although MOF metamodels still lack a precise semantics, they provide a semi-formal and easy-to-understand language for describing the structure of system and mathematical models uniformly. As only a small (but still meaningful) subset of MOF constructs are used for our purpose, we expect that a future (mathematically precise) metamodelling approach will contain all these features (valuable foundations are [26, 30]).

1.3 Theoretical Foundations of Model Transformation

In this section, basic concepts of graph transformation systems (such as graphs, graph transformation rules, transformation units, etc.) are applied to the special needs of model transformation built upon MOF metamodels in order to provide a precise (but practice oriented) mathematical background. For the basic definitions, we will basically follow the directions of [1], while further details on the theoretical foundation of model transformations can be found in [29].

1.3.1 Graph Models

Definition 1.3.1 *A **directed graph** $\mathsf{G} = (\mathsf{NODES}, \mathsf{EDGES}, \mathsf{source}, \mathsf{target}, \mathsf{label})$ consists of a finite set NODES of **nodes** , a finite set EDGES of **edges**, two **mappings** assigning the source and the target node to each edge, and a mapping label, assigning a labelling symbol from a given alphabet to each node and each edge.*

*An edge e in G goes from the source $\mathsf{source}(\mathsf{e})$ to the target $\mathsf{target}(\mathsf{e})$ and is **incident** to $\mathsf{source}(\mathsf{e})$ and $\mathsf{target}(\mathsf{e})$.*

Definition 1.3.2 *A graph L is a **subgraph** of G, denoted by $\mathsf{L} \subseteq \mathsf{G}$, if the node and the edge sets of L are subsets of the respective sets of G, and the source, the target and label mappings of L coincide with the respective mappings of G restricted to L.*

Definition 1.3.3 *L has an **occurrence** in G, denoted by $\mathsf{L} \to \mathsf{G}$, if there is a mapping occ which maps the nodes and the edges of L to the nodes and the edges of G, respectively, and preserves sources, targets, and labellings.*

Definition 1.3.4 *Labels in **typed graphs** are divided into classes, called types, and that edges of a certain type are restricted to be incident only to certain types of source and target nodes. Typed graphs can be specified by so–called **graph schemata** as e.g. in [23].*

Definition 1.3.5 ***Attributed graphs** are equipped with attributes. Attributes can be of different types (a number, a text, an expression, a list or even a graph).*

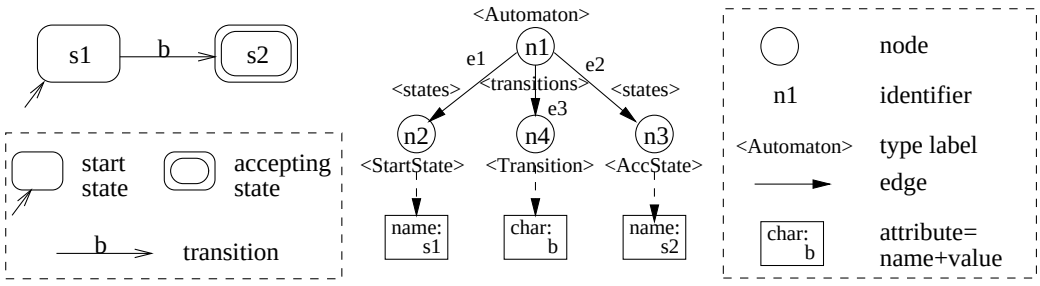
Definition 1.3.6 *A **model graph** G is a directed, typed and attributed graph with the following structure.*

- *A graph node is associated with a unique identifier Id, and a type label $\mathsf{T_n}$.*
- *An edge has an own Id, a reference to a source $\mathsf{Id_S}$ and a target $\mathsf{Id_T}$ identifier, and a type label $\mathsf{T_e}$.*
- *Both nodes and edges may be related to attributes (represented e.g. as special graph nodes) with an Id identifier (referring to the graph element the attribute is related to), a type label $\mathsf{T_a}$ and a data value V.*

Model graphs may also contain n-ary relations between nodes (denoted as relations or hyper-edges), but these relations are represented by a special class of model graph nodes connected to "original" graph nodes by special (reserved) types of edges. Model graph relations are closely related (in their use and functionality) to PROGRES path expressions [24] with the extension of having more than one source and/or target nodes.

The concepts of a model graph were introduced in order to obtain a close correlation with MOF based models. In a model graph, each node and edge must have type labels corresponding to a MOF construct in the metamodel. This correspondence is characterized by the following rules:

- Instances of a MOF Class (A) are mapped into model graph nodes with identically named types.
- Instances of a *navigable* MOF AssociationEnd (E) between two MOF Classes (from A to B) are projected into model graph edges with the further type restriction that all the graph edges of type E have to connect a graph node of type A to a node of type B. (As a result, a MOF Association with two navigable AssociationEnds are projected into two separate directed model graph edges).
- MOF Attributes are directly mapped into model graph attributes.



(a) Finite automaton (b) Model graph of FA

Figure 1.5: Model graphs from MOF based models

Example 1.3.7 In this respect, the model graph of Figure 1.5(b) is an equivalent of the simple automaton of Figure 1.5(a). The model graph contains

- **nodes** such as `n1`, `n2`, `n3` and `n4` (in the following, we refer to a graph object by its identifier) of type `Automaton`, `StartState`, `AccState` and `Transition` respectively.
- **edges** like `e1` and `e2` of type `states` and `e3` of type `transitions` connecting the automaton node with its states and transitions.
- **attributes** attached to each node that contain the name and the value of the attribute such as `name: s1` for states and `char: b` for the transition.

.

Reference graphs As the main goal of model transformation is to derive a target model from a given source model, source and target objects must be linked to each other in some way to form a single graph. For this reason, the following definition introduces the concepts of a **reference graph**. The structure of a reference graph is also constrained by a corresponding reference metamodel, which contains (i) references of existing source and target metamodel nodes; (ii) novel (so–called) reference nodes that provide a typed coupling of source and target objects, and (iii) reference edges connecting all these nodes.

Definition 1.3.8 *A **reference graph** $G_{ref} = (G_s, G_t, NODES_{ref}, EDGES_{ref})$ contains a source and a target model graph (G_s and G_t respectively), and an additional set of reference nodes $NODES_{ref}$ and edges $EDGES_{ref}$, where*

- *a **reference node** is a model graph node (thus associated with a unique identifier Id, and a type label T_n) of the reference metamodel. Reference nodes will be depicted by rounded boxes in the sequel for being able to distinguish them from source and target objects.*
- *a **reference edge** is a model graph edge (of the reference metamodel) that may lead* from a reference node *to either a source, a target or a reference element of a specific type.*

The previous definition of reference graphs can be extended to refer not only to single nodes but to a subgraph of source and target models. As a result, a more complex and refined reference structure is obtained. However, the reference graph is no longer a simple graph but a hierarchical graph (where nodes can be refined in turn to entire graphs).

In the following section, operations (graph transformation rules) will be defined that would manipulate an arbitrary class of graphs (including, naturally, model and reference graphs).

1.3.2 Transformation rules

Graph transformation consists of applying a rule and iterating this process. Each **rule application** transforms a graph by replacing a part of it by another graph (which is conceptually similar to textual pattern manipulation in Chomsky grammars).

Definition 1.3.9 *A **graph transformation rule** $r = (L, R, Emb, App)$ contains a left–hand side (LHS) graph L, a right–hand side (RHS) graph R, some embedding mechanisms Emb and application conditions App.*

Definition 1.3.10 *The **application** of r to a **host graph** (graph instance) G replaces an occurrence of the LHS L in G by the RHS R. This is performed by*

1. *finding an occurrence of L in G (also denoted as graph pattern matching),*
2. *checking the application conditions App (e.g negative application conditions which prohibit the presence of certain nodes and edges)*
3. *removing a part of the graph G determined by the occurrence of L yielding the **context graph** D,*
4. *gluing R and the context graph D by using the embedding mechanism Emb, and obtaining the **derived** graph H.*

The computational complexity of graph transformation depends mainly on the complexity of graph pattern matching. Unfortunately, this problem is equivalent with the subgraph isomorphism problem, which is known to be NP-complete. However, existing graph pattern matching approaches (e.g. search paths in PROGRES [**?**], or the constraint satisfaction method [15]) show an acceptable run–time behaviour for practical applications.

The previous definition covers several graph transformation approaches, each of them answering the following two main questions concerning rule application differently.

- Shall we allow that the match of a rule is a non-isomorphic image of its LHS, i.e. that two nodes of the LHS share the same node in the host graph?
- If deletion of a node is ordered by a rule, how to handle (or avoid) dangling edges, i.e. those edges that are connected to the node to be deleted but not handled by the specific rule (as a graph must be obtained after each derivation step).

The aim of model transformation is to generate a target model from scratch that is semantically equivalent with a given source model. For this reason, each occurrence of a specific LHS pattern has to be transformed according to the RHS. Moreover, the majority of model transformation rules are non–deleting, which ensures the pleasant property of being able to handle all the LHS matches parallelly (parallel execution).

On the other hand, when the deletion of certain graph objects is prescribed by a rule, we must ensure that distinct parallel matches do not confront with each other. In our model transformation approach, parallelly executable rules cannot remove any part of the graph to avoid such problems.

Following the classification of different graph transformation approaches that can be found in [22], a model transformation rule is defined by answering the previous questions as follows.

Definition 1.3.11 *A **model transformation rule*** r_{mt} *is a special graph transformation rule, where*

- *both graphs* L *and* R *are reference graphs;*
- *an occurrence of* L *in* G_{ref} *is not required to be an isomorphic image of* L *(two nodes in the LHS may share the same node in the host graph);*
- *all the dangling edges are deleted automatically (as deletion is a rare operation);*
- *if the LHS graph is composed of more than one components than all but one component must contain a node obtained as parameter*

To limit the worst-case complexity of the graph pattern matching in model transformation rules, the last requirement in the definition necessitates that if the LHS graph of a rule is constructed from more components then a node of each component (except for one) has to be identified by input parameters passed to the rule. In this way, having more components in the LHS will not drastically decrease run-time performance.

For practical applications, model transformation rules are allowed to contain parameter nodes and edges. An input parameter is mapped to the LHS of the rule, and determines the image of the parameter node (or edge) in constant time (supposing that its occurrence was determined beforehand). An output parameter requires a RHS object to be mapped to it.

Fortunately, the theoretical bases of model transformation rules need not be altered as input and output parameters merely provide additional application conditions. Thus, we may add parameters to transformation rules in the sequel.

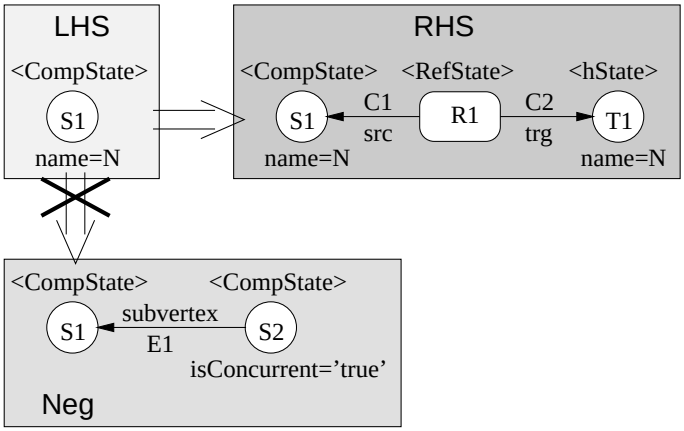


Figure 1.6: The structure of model transformation rules

Example 1.3.12 A sample model transformation rule is depicted in Figure 1.6. It consists of a LHS, a negative application condition (Neg) and a RHS graph.

- The reference graph of the LHS contains a single node of type `CompState` identified by the variable `S1` and with a single attribute called `name` having a value `V` (all the identifiers and attribute values that begin with capital letters are identifiers while data values are printed between quotation marks).

- The rule contains a negative condition prescribing that node `s1` must not be connected to a concurrent `CompState` node by a `subvertex` edge (attributes are indicated this time without boxes). The mapping between LHS and Neg nodes and edges are described by identically named identifiers (such as `S1`). Such a mapping prescribes that the nodes in the LHS and Neg must share the same instance in the host graph.

- The RHS of the rule contains the `S1` node (note the mapping with identical names), two additional nodes `R1` (a reference node of type `RefState`) and `T1` (a target node of type `hState`) connected by edges `C1` and `C2` (of type `src` and `trg`).

As the value `N` of attribute `name` in `T1` can also be mapped to the attribute of `S1` (such a mapping is also indicated by a Prolog like unification), the values of the two attributes are identical. In a more general case, the value of a newly constructed attribute may be defined by a corresponding function. In this way, we may transform numerical values in addition to graph structure.

Example 1.3.13 The *parallel application* of the previous rule is demonstrated in Figure 1.7.

- The reference host graph to which the rule is planned to be applied is depicted in Figure 1.7(a). Let `CompState` nodes without an `isConcurrent` attribute denote the default case when this value is false.

- The first step (Figure 1.7(b)) is to find an occurrence of the LHS in the host graph, i.e. a node of type `CompState`. In the current host graph, the LHS pattern can be matched three times (`s1`, `s2` and `S4`). The node `s1` can be matched because the LHS pattern prescribes no conditions for the `isConcurrent` attribute. The node `s3` cannot be matched as its

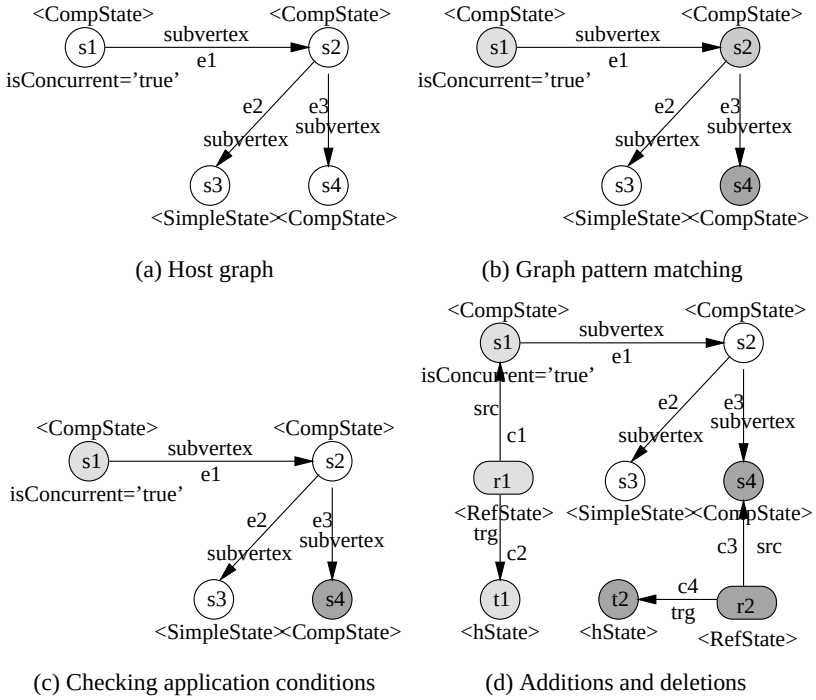


Figure 1.7: Applying model transformation rules

type `SimpleState` does not correspond to the type of the pattern node (`CompState`). As we apply our rule parallelly (as the default semantics for rule application), each parallel manipulation process is indicated by different colours.

- Afterwards, the negative application condition is checked, which prescribes that the LHS node `S1` must not be connected to a concurrent `CompState` node `S2` by a `subvertex` edge. At this point, the occurrence of node `s2` is invalidated as it can be extended by the pattern of the Neg graph (a `subvertex` edge leading from a concurrent `CompState`). On the other hand, the occurrence of `s1` is valid as there are no edges of type `subvertex` in the host graph that lead to `s1`.

- At third, the occurrences of those rule graph objects that appear only on the LHS but not on the RHS are deleted from the host graph. In our sample rule, no such deletions are needed to be performed.

- Finally, those objects that appear only on the RHS (but not on the LHS) are added to the host graph. In our model transformation rule, a reference node and a target node connected by corresponding edges should be created for each matches. For this specific application, references nodes `r1` and `r2` of type `RefNode` are created together with target nodes `t1` and `t2` of type `hState` and reference edges `c1`, `c2`, `c3` and `c4`.

1.3.3 Transformation Units

As possible industrial applications of model transformation surely consist of very large and complex models containing hundreds of rules, model transformation rules must be extended by a sophisticated structuring mechanisms that allow to compose them in a modular way. In the graph

transformation community, the concepts of *transformation units* were introduced for such purpose (e.g. [1, 13]).

Definition 1.3.14 *A **transformation unit*** $\mathsf{tu} = (\mathsf{I}, \mathsf{U}, \mathsf{R}, \mathsf{C}, \mathsf{T})$ *is a system where* I *and* T *are graph class expressions (describing initial and terminal graphs),* R *is a finite set of rules and* C *is a control condition, and* U *is the set of imported transformation units (which is empty, initially).*

Definition 1.3.15 *A **model transformation unit** is a transformation unit where*

- *the **graph model** conforms to those reference graphs described in Definition 1.3.8;*
- *the set* R *of **rules** is well–formed* model transformation rules,
- *the class of **control conditions*** C *(containing control flow information) is composed of* extended regular expressions *(discussed in details e.g. in [14]). Each* $\mathsf{c_i}$ *is either a transformation unit or rule identifier or a previously defined control condition.*
  - skip *is the idle operation*
  - $\mathsf{try(c)}$ *applies* c ***at most once** (once if* c *is applicable and fails otherwise); This operation is also denoted as **test**;*
  - $\mathsf{forall(c)}$ *represents the **parallel application** of* c*;*
  - $\mathsf{c_1}, \mathsf{c_2}$ *stands for the control flow in which* $\mathsf{c_2}$ *is applied right after* $\mathsf{c_1}$ *(**sequence**);*
  - $\mathsf{c_1}; \mathsf{c_2}$ *represents such a control flow where we choose from* $\mathsf{c_1}$ *and* $\mathsf{c_2}$ *non–deterministically (**choice**);*
  - $\mathsf{if\ c\ then\ c_1\ else\ c_2}$ *serves as a **branch** of the control flow (depending on the evaluation of* c*);*
  - $\mathsf{c!}$ *applies* c ***as long as possible**;*
  - $\mathsf{c_1}|\mathsf{c_2}$ *stands for such a control flow where* $\mathsf{c_1}$ *and* $\mathsf{c_2}$ *can be applied parallelly (**fork**).*

Initial and terminal class expressions serve as preconditions and postconditions on graphs transformed by the unit. Model transformation rules are nested into transformation units, which units themselves can be imported by further transformation units (circular import is usually forbidden). In this sense, the entire transformation is defined in a hierarchical way; similarly to the process of IT system design.

Several ways of non–determinism are embedded in the application of graph transformation rules, for instance choosing an appropriate rule to be applied or finding an occurrence of the LHS of the rule in the graph. Although non–determinism is often useful in the phase of a mathematical analysis, it has to be eliminated in practical applications before the implementation phase. Control conditions provide a natural mechanism to restrict the control flow of model transformation.

Example 1.3.16 Figure 1.8. shows a sample model transformation unit (`sampleTU`), which derives the graph $\mathcal{G}'$ from input graph $\mathcal{G}$. The transformation unit states that

- the initial and terminal graph must be a well–formed reference graphs,
- `variantTU` has three rules called `variantR`, `adjudR` and `distVoteR`
- two further units (not discussed here in details) are imported, namely, `ftsTU` and `linkTU`
- the control condition prescribes that

sampleTU($\mathcal{G}, \mathcal{G}'$)*:*
initial: `reference_graph`($\mathcal{G}$)
terminal: `reference_graph`($\mathcal{G}'$)
rules: `variantR, adjudR, distVoteR`
uses: `ftsTU,linkTU`
control: `ftsTU,(((variantR!),linkTU)` | (**if** c **then** `adjudR` **else** `distVoteR`))

Figure 1.8: Model transformation unit "variantTU"

1. `ftsTU` is executed first;
2. the control flow forks afterwards;
   (a) in one thread we should apply `variantR` as long as possible followed by the transformation described in `linkTU`;
   (b) in the other thread, if condition `c` evaluates to true then `adjudR` is applied otherwise `distVoteR` is executed.

1.4 Benchmark Transformation

In the previous section, a theoretical framework of model transformations has been introduced which framework has also provided a close correspondence between MOF based models and graph transformation rules. In the sequel, the concepts of model transformation will be illustrated on a benchmark example which transforms UML statecharts into their extended hierarchical automata equivalents.

This section will be organized to follow the scenario of model transformations discussed in Section 1.1.4. Both source and target models are introduced informally at first, which descriptions are followed by a more detailed specification of their structural semantics by the corresponding MOF metamodels.

1.4.1 An Informal Introduction to UML Statecharts

UML statecharts are an object–oriented variant of classical Harel statecharts [10] and they are a notation for describing behavioural aspects of the system under design. In fact, the statechart formalism itself is an extension of traditional state transition diagrams.

Figure 1.9 summarizes the basic notation of UML statecharts, while the sample statechart of Figure 1.10 will serve as the source model of the transformation later on.

States Statecharts are basically constructed from states and transitions. In fact, one of the main concepts of statecharts is state refinement. In Fig. 1.10 state `s1` is refined into two distinct automaton (represented as a single state), `s4` and `s5`, each of them is refined in turn into an automaton consisting of further substates (e.g. `s6`, `s7`). States refined to sub–states are denoted as **composite**, additionally, `s4` and `s5` are called **concurrent** substates of `s1`.

"System states" are modelled by **configurations**, which are sets of active states. For instance, our sample system can be any of the following configurations: `{s1,s6,s8}`, `{s1,s6,s9}`, `{s1,s7,s8}`, `{s1,s7,s9}`, `{s2}`, `{s3}`.

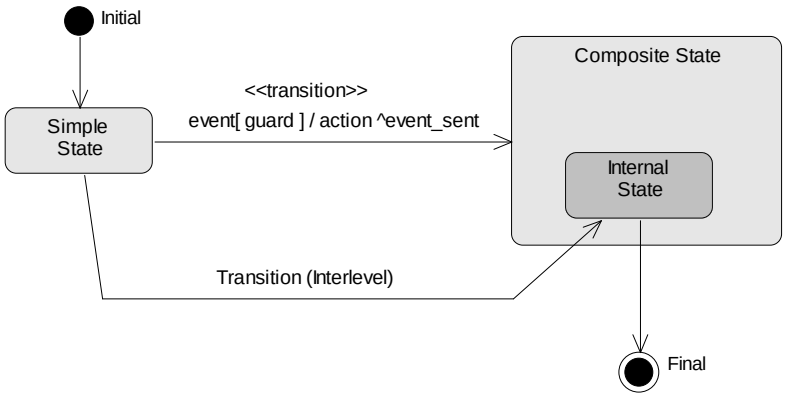


Figure 1.9: UML Statechart notation

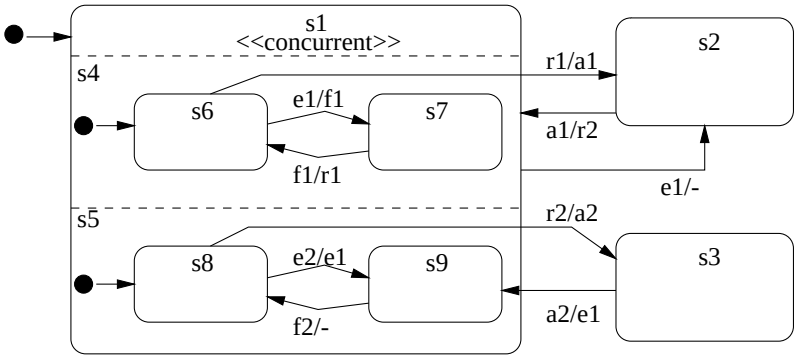


Figure 1.10: The source Statechart model

Transitions A transition connects a **source** state to a **target** state. A transition is labelled by a trigger **event**, a boolean **guard** and a sequence of **actions**.

A transition is enabled and can fire if and only if its source state is in the current configuration, its trigger is offered by the external environment and the guard is satisfied. In this case, the source state is left, the actions are executed, and the target state is entered.

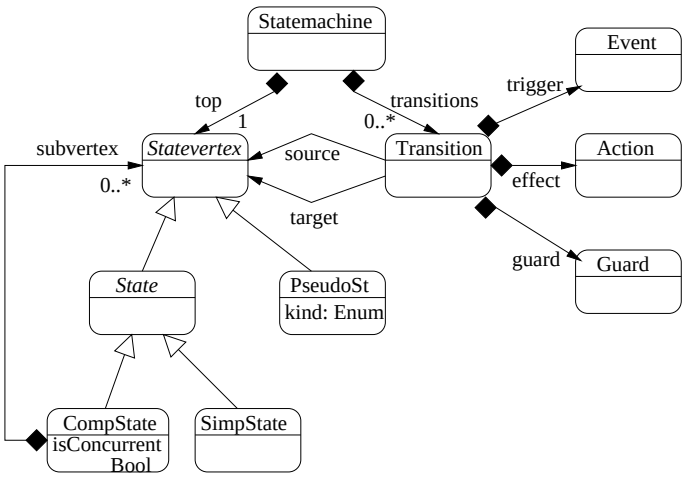
In our example, if event `a1` is offered and the current configuration is `{s2}`, state `s2` is left and state `s1` is entered. In particular, as `s1` is composite, we also have to define which are the *substates* that are reached. In the case at hand, they are the default ones specified by the **initial** states of `s4` and `s5`, namely, `s6` and `s8`. In a general case, the source and target state of a transition may be at a different level of the state hierarchy. Such a transition is denoted then as **interlevel**.

Event dispatching In general, more than one event can be available in the environment. The UML semantics assumes a **dispatcher**, which selects one event at a time from the environment and offers it to the state machine. As a result, more than one transition can be enabled, which may cause a conflict to be resolved if the intersection of the states left by the enabled transitions is not empty. Conflicting transitions are tried to be resolved by using priorities: a transition has higher priority than another transition if its source state is a substate of the other transition's source state. If conflicts cannot be resolved by priorities, any of the enabled transitions can be fired, moreover,

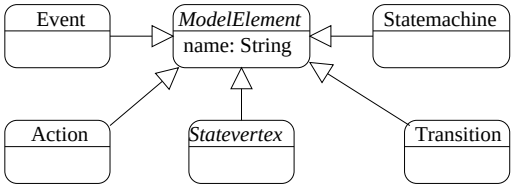
in case of concurrent states, more than one transition is fired at a time.

1.4.2 The Metamodel of UML Statecharts

In the following, the major features and components of *UML statecharts* will be described by means of the standard MOF–based UML metamodel (Figure 1.11). However, additional semantic restrictions expressed in OCL are omitted this time due to the lack of space.



(a) Core concepts



(b) Inheritance from ModelElement

Figure 1.11: The metamodel UML Statecharts

- The top-level class is called **Model Element**. It is an abstract superclass (thus without instances) with a single attribute `name`.
- A **state machine** is a behaviour that specifies the sequences of states that an object goes through during its life in response to events, together with its response and actions.

  In the metamodel a `StateMachine` is composed of a `top` State and an arbitrary number of `transitions`.

  - The `top` association defines the top level `State` (exactly one as depicted by its multiplicity `1`) directly owned by `StateMachine`. Further `States` are owned by parent composite states and discussed later.
  - The `transitions` role relate the `StateMachine` to its `Transitions`. All `Transitions` are owned directly by at most one `StateMachine`.

- A **state vertex** is an abstract class in the statechart. In general, it can be the source or destination of any number of transitions.

  In the metamodel a `StateVertex` is a subclass of `ModelElement`.

- A **state** is a condition or situation during the life of an object meanwhile it satisfies some condition, performs some action or waits for some events.

  In the metamodel `State` is an abstract class and a subclass of `StateVertex`.

- A **simple state** is a state that does not have substates.

  In the metamodel a `SimpleState` is a subclass of `State` and it does not have any additional features.

- A **composite state** is a state that consists of substates.

  In the metamodel a `CompositeState` is a subclass of `State`.

  - Its `subvertex` association denotes a set of `States` that form the substates of a `CompositeState`. Each substate is uniquely owned by its parent `CompositeState`, and self–containment is not allowed either.
  - The `isConcurrent` attribute has a boolean value that specifies the decomposition semantics: if this attribute is true, then the composite state is decomposed directly into two or more orthogonal conjunctive regions (usually associated with concurrent execution). Otherwise, there are no direct orthogonal region in the composite state. This means that exactly one of the substates can be active at a given instant (i.e. sequential execution).

- A **pseudo state** is an abstraction of different types of nodes including initial and final states.

  In the metamodel a `PseudoState` is a subclass of `StateVertex`. It possess the `kind` attribute that can be e.g. **initial**, **final**, **fork** or **branch**.

  An additional semantic constraint here should state that each non-concurrent composite state must have exactly one initial pseudo state.

- A **transition** is a binary relationship between a `source` state vertex and a `target` state vertex.

  In the metamodel`Transition` is a subclass of `ModelElement` that participates in various relationships with other state machine metaclasses (by associations):

  - `trigger` specifies the single `Event` which activates it
  - `guard` is a predicate that must evaluate to true at the instant the transition is triggered;
  - `effect` specifies an `Action` which has to be performed after the transition is fired.
  - `source` denotes the `StateVertex` affected by firing the `Transition`.
  - `target` denotes the `StateVertex` that results from a firing of the `Transition` when the `StateMachine` was originally in the source `State`. After the firing the `StateMachine` is in the target `State`.

- An **event** may lead to the activation of a some internal behaviour in an object.

  In the metamodel an `Event` is a subclass of `ModelElement` and is a part of a `Transition` by representing its `trigger`.

- A **guard** condition is a boolean expression that may be attached to a transition in order to determine whether that transition is enabled or not.

  In the metamodel `Guard` is a `ModelElement`. Its `expression` attribute is a boolean expression which specifies the guard condition.

- An **action** may also lead to the activation of a some internal behaviour in an object after a transition has been fired.

  In the metamodel an `Event` is a subclass of `ModelElement` and is a part of a `Transition` by representing its `effect`.

Please note that the UML metamodel was slightly simplified (with respect to the standard UML metamodel) in order to keep the level of legibility of the paper. In our opinion, these modifications do not have major impact on statechart semantics.

1.4.3 An Introduction to Extended Hierarchical Automaton

The concepts of Extended Hierarchical Automaton (EHA) were introduced in [17] for the first time. However, in the current paper a slightly modified version is used (following [16]) since the latter papers contain the textual descriptions of the Statechart–EHA mapping which serve as a basis of our benchmark model transformation.

Extended Hierarchical Automaton provide a formal operational semantics for UML Statechart diagrams. Having only a small number of rules (see [16] for details on rules) facilitates the formal proof of properties that show the correctness of the formal semantics with respect to the requirements formulated in the definition of UML[20].

However, the current paper the discussion of these operational semantic aspects is omitted as the transformation between the SC and EHA notation is a structural one (according to [16]). In other words, we transform a statechart structure into an EHA structure, which contains all the basic information that is needed for specifying its semantic behaviour.

Definition 1.4.1 (Sequential Automaton) *A sequential automaton* A *is a 4-tuple* $(\sigma_\mathsf{A}, \mathsf{s}^0_\mathsf{A}, \lambda_\mathsf{A}, \delta_\mathsf{A})$ *where* σ_A *is a finite set of* ***states*** *with* $\mathsf{s}^0_\mathsf{A} \in \sigma_\mathsf{A}$ *the* ***initial state****,* λ_A *is a finite set of* ***transition labels*** *(labels have a particular structure) and* $\delta_\mathsf{A} \subset \sigma_\mathsf{A} \times \lambda_\mathsf{A} \times \sigma_\mathsf{A}$ *is the* ***transition relation****.*

Definition 1.4.2 (Transition Labels) *A transition label* $\lambda \in \lambda_\mathsf{A}$ *is a 5-tuple* $(\mathsf{sr}, \mathsf{ev}, \mathsf{g}, \mathsf{ac}, \mathsf{td})$*, where* sr *is the* ***source restriction****,* ev *is the* ***trigger*** *event,* g *is the* ***guard****,* ac *is the list of actions, while* td *is the target determinator.*

Definition 1.4.3 (Hierarchical Automaton) *A hierarchical automaton* H *is a 3-tuple* $(\mathsf{F}, \mathsf{E}, \rho)$*, where* F *is a finite set of sequential automaton with mutually disjoint sets of states,* E *is a finite set of* ***events****, and the* ***refinement function*** $\rho : \bigcup_{\mathsf{A} \in \mathsf{F}} \sigma_\mathsf{A} \rightarrow 2^\mathsf{F}$ *imposes a tree structure to* F*.*

Example 1.4.4 A sample hierarchical automaton (which will turn to be the alternate representation of the statechart in Figure 1.10) is depicted in Figure 1.12 with a self-explanatory notation. The transition labels are listed in Table 1.2.

Please note that there are no “interlevel” transitions in the EHA model. The source and target states of the original SC transition (see `t5`, for instance) are encoded into transition labels (`s6`, `s9`).

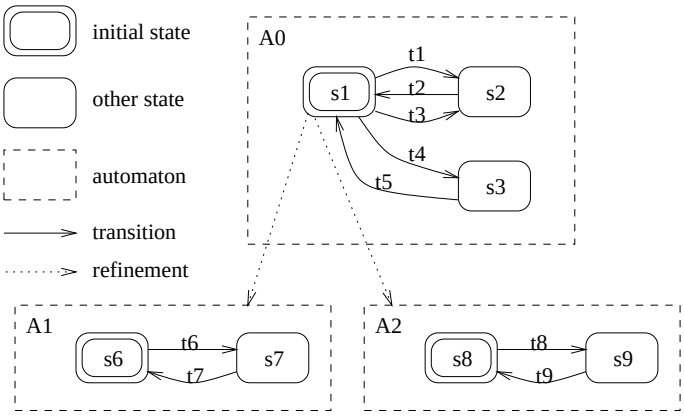


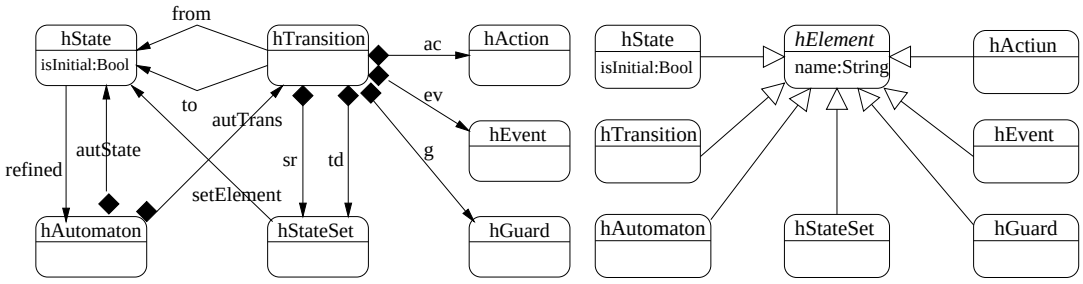
Figure 1.12: A sample Extended Hierarchical Automaton (EHA)

t	t1	t2	t3	t4	t5	t6	t7	t8	t9
SR t	{s6}	$\emptyset$	$\emptyset$	{s8}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
EV t	r1	a1	e1	r2	a2	e1	f1	e2	f2
AC t	a1	r2	ϵ	a2	e1	f1	r1	e1	ϵ
TD t	$\emptyset$	{s6,s8}	$\emptyset$	$\emptyset$	{s6,s9}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 1.2: Transition labels

1.4.4 The Metamodel of Extended Hierarchical Automaton

According to the previous formal definitions, the MOF metamodel of EHA will be constructed in the sequel (Figure 1.13).



(a) Core concepts (b) Inheritance from hElement

Figure 1.13: The metamodel of Extended Hierarchical Automaton

- The MOF Class `hElement` is the top–level abstract Class of the EHA metamodel. It has a single attribute `name` which contains the labels for states and transitions.
- A `hAutomaton` specifies an EHA sequential automaton. It is a subclass of `hElement`, and has two navigable AssociationEnds;
  - `autState` accesses the arbitrary number of states (`hState`) of the automaton (at

the current level). A state belongs to exactly one automaton as the set of states are disjoint.

– `autTrans` lists all the transitions (`hTransition`) that belong to the specific sequential automaton. A transition belong to exactly one automaton this time as well.

- The MOF Class `hState` denotes the states of EHA. It is a subclass of the top–level `hElement`.
  - The attribute `isInitial` specifies whether the state is an initial state of an automaton. A semantic constraint here should prescribe that each automaton may not have more than one initial state.
  - If an EHA state is refined to one or more sub–automatons, these automatons are accessible via the AssociationEnd `refined`. An automaton is a refinement of exactly one state (e.g. `A0` in Fig. 1.12 is a refinement of the top–level state which is not depicted explicitly).
- The MOF Class `hTransition` is the EHA equivalent of UML transitions. It is a subclass of `hElement`, and possesses the following navigable AssociationEnds.
  - `from` specifies the source state of an EHA transition (please note that only non–interlevel transition are allowed in EHA according to the definition). A transition has exactly one referred source state and a state may be the source of an arbitrary number of transitions.
  - `to` refers to the target state of a transition (the rest of its semantics is similar to one of `from`).
  - `ev` is for determining the event that has triggered the specific transition.
  - `g` describes the logical guard condition of the transition
  - `ac` is used for accessing the corresponding action that the triggered transition has executed.
  - The source restriction and target determinator of a transition (called `sr` and `td` respectively) is composed of two collector Classes of type `hStateSet`.
- The Class `hAction` is a simplified EHA representation for UML Actions.
- The MOF Class `hEvent` corresponds to the UML Event construct.
- The MOF Class `hGuard` corresponds to the UML Guard class.
- The instances of a `hStateSet` Class are related (by the association `setElement`) to an arbitrary number of `hStates`, while a `hState` may belong to more than a single `hStateSet`.

After having introduced the structure of source and target models, we turn on to their transformation. As a starting point, our goals could be summarized as taking the source SC model of Figure 1.10. as input and obtaining the EHA model of Figure 1.12 as the result.

1.4.5 An Informal Description of the Statechart–EHA Transformation

The model transformation mapping UML statecharts into their EHA equivalents will be sketched informally according to [16]. An Extended Hierarchical Automaton $\mathsf{H} = (\mathsf{F}, \mathsf{E}, \rho)$ will be defined by the set of sequential automata F, the refinement function ρ and a set of events E.

Set of sequential automata Each automaton $\mathsf{A} \in \mathsf{F}, \mathsf{A} = (\sigma_\mathsf{A}, \mathsf{s}^0_\mathsf{A}, \lambda_\mathsf{A}, \delta_\mathsf{A})$ is defined as follows.

- **States**. States of the statechart are uniquely mapped to states of sequential automata.
  - *Root automaton* H. If the (composite) top state s_0 of the statechart is concurrent then it is mapped to the single (initial) state of a degenerate root automaton H. Otherwise, the direct substates of the top state are mapped to states σ_H of the root automaton H. (In practical applications, the top–state is never concurrent.)
  - *Each sub–automata of* H. Each non–concurrent composite substate s defines the states of a unique sequential automaton A_s, as direct substates of s are mapped to states of $\sigma_{\mathsf{A}_\mathsf{s}}$. Note that regions (direct substates of a concurrent composite state) are not mapped to any state in the extended hierarchical automaton.
- **Initial state**. The initial state s^0_A of an automaton A is the state that corresponds to the state of the statechart marked by an initial pseudo state.
- **Transitions**. In order to define the mapping of the transitions, we need the following definitions. A transition of the statechart is characterized by its lowest common ancestor (LCA) state, which is the *lowest level non–concurrent* state that contains all the source and target states. The *main source* (*main target*) of a transition is the direct substate of its LCA that contains the sources (targets). According to the above rules, main sources and main targets are always transformed to states of the same automaton.

  Each transition τ in the statechart is mapped to a unique transition t of the EHA as follows. The *source* (*target*) of t is the state that corresponds to the main source (main target) of τ. This means that a compound or interlevel transition of the statechart is mapped to a transition of the automaton containing the states corresponding to its main source and main target (this automaton is a sub–automaton of the state representing the LCA). The original source and target states will be included in the label of the transition in the form of source restriction and target determinator as described below.
- **Transition labels**. The label of a transition t is of the form $(\mathsf{SR\ t}, \mathsf{EV\ t}, \mathsf{G\ t}, \mathsf{AC\ t}, \mathsf{TD\ t})$ where $\mathsf{SR\ t}$ and $\mathsf{TD\ t}$ are generated using the source(s) and target(s) of τ, while $\mathsf{EV\ t}$, $\mathsf{G\ t}$ and $\mathsf{AC\ t}$ of t are inherited from τ:
  - *Source restriction.* If the set of states that correspond to the source(s) of τ is the same as the source of t, then $\mathsf{SR\ t}$ must be empty, otherwise it is the corresponding set of sources.
  - *Target restriction.* $\mathsf{TD\ t}$ is the normalized set of states that corresponds to the target(s) of τ. Normalizing means computing the maximal set of orthogonal basic states that are substates of the states entered by τ explicitly or by default. In this way, $\mathsf{TD\ t}$ explicitly contains all the states which have to be entered when the transition is fired, while some of these states are not explicitly pointed by τ. The following is a sketch of a normalization algorithm which visits the states reached by (segments of) τ, starting from its main target:
    - If a basic state is reached then it is added to $\mathsf{TD\ t}$ and the recursion stops.
    - If a composite state is reached at its boundary then the algorithm is applied recursively to its initial substate, or to the initial substate of each of its regions.

  - If a non–concurrent composite state is reached and its boundary is crossed then the algorithm is applied recursively to its direct substate where the transition continues.
  - If a concurrent composite state is reached and its boundary is crossed then the algorithm is applied recursively to (i) the direct substates of those regions where the transition continues and (ii) the initial substates of the other regions.
- *Trigger events.* In UML statecharts, each transition can have at most one trigger event, since join, fork, and branch segments cannot have a trigger. Accordingly, $\mathsf{EV}\ \mathsf{t}$ is equal to the trigger event of τ.
- *Guards.* Each transition may have a single guard; accordingly $\mathsf{G}\ \mathsf{t}$ is exactly the guard of τ.
- *Actions.* $\mathsf{AC}\ \mathsf{t}$ is exactly the sequence of actions of τ.

- **Refinement function**. ρ is determined by the subvertex relationships of composite states. If a composite state s is non-concurrent and it is not a region then its direct substates form the states of $\mathsf{A_s}$, a sub–automaton of s, where $\{\mathsf{A_s}\} = (\rho\ \mathsf{s})$. If a composite state s is concurrent then each of its regions forms a sub–automaton of s, in such a way that this automaton contains the direct substates of the region.

- **Set of events**. E is defined as the union of two (not necessarily distinct) sets: the set of events used in the statechart as triggers of the transitions and the set of events generated by actions. In open systems, the set of events generated by the environment is also included.

1.4.6 Reference metamodel

Before being able to specify the SC-EHA model transformation by means of graph transformation rules, the reference structure between the two models also has to be defined by a corresponding MOF metamodel. In the current paper, we have chosen a simple metamodel for references, however, arbitrarily complex types can be introduced for reference nodes and edges.

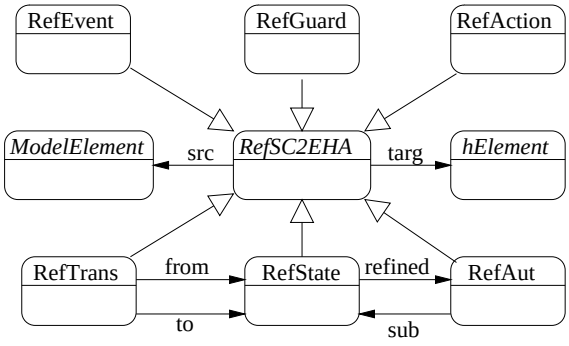


Figure 1.14: The reference metamodel of the SC–EHA transformation

- In general, a reference node relates a SC node to an EHA node by edges `src` and `trg`, as indicated by the abstract superclass `RefSC2EHA` of the reference metamodel.

- A reference node is introduced for each target class (except for `hStateSet`) by inheriting all the properties from the abstract class `RefSC2EHA`.

- Further reference edges (such as `from` or `refined`) correspond to EHA associations

Although the transformation rules themselves will express further connection constraints on specific types of reference nodes (like e.g. a `RefState` will always relate a `CompState` or a `SimpleState` source node to a `hState` target node), these constraints are not expressed in the structure of the reference metamodel.

Now we have a metamodel for the source, the target and the reference model. The graphs in model transformation rules (e.g. LHS or RHS) will be constructed by using only those nodes and edges that are allowed according to these metamodels. In other words, the rule graphs themselves have to be well–formed reference graphs.

1.5 Formal Specification of the SC-EHA Transformation

In the current section, the SC-EHA transformation will be specified formally by means of model transformation rules and units constructed according to the previous informal specification.

The scenario of the SC–EHA model transformation is the following. The transformation starts with projecting SC states into EHA hStates and hAutomaton. Then hStates are related to their hAutomaton at a second phase. Afterwards, the transitions of the statecharts are handled by a set of transformation rules (including the trivial transformation of Actions, Events and Guards), also introducing several auxiliary rules for preprocessing the SC model. [1].

Due to the lack of space, the rules for creating source restrictions and target determinators are omitted from the current paper. However, all the necessary relations (least common ancestor, main source and target) will be defined, thus the specification of these rules may serve as graph programming exercises for the interesting reader.

1.5.1 Transforming States

Transforming simple states At first (rule `simpleStateR` in Figure 1.15), each source node `S1` of type `SimpleState` is transformed into a corresponding `hState` node by adding a target node `T1`, a reference node of type `RefState` and the connecting `src` and `trg` reference edges.

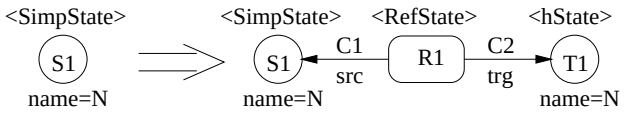


Figure 1.15: The `simpleStateR` rule

Transforming composite states Composite states of a UML statechart are also transformed into EHA hStates (Figure 1.16). According to Section 1.4.5, states representing the regions of a concurrent state are not projected into the states of the EHA automaton. This fact is expressed by a negative application condition prohibiting the presence of a parent concurrent composite state `S2` for the current LHS match. The RHS prescribes additions similar to `simpleStateR`.

[1] Please note that in the current section, each transformation rule has to be executed parallelly as default if not stated otherwise.

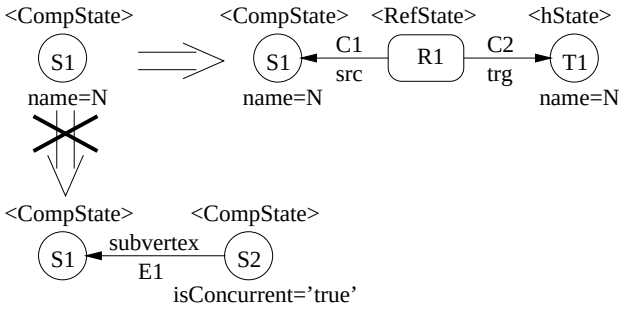


Figure 1.16: The `compositeStateR` rule

Marking the initial states In the EHA metamodel, the initial states of each automaton are marked by setting the `isInitial` attribute of the hState to true. Therefore, rule `initStateR` (Figure 1.17) should match all the simple and composite SC states (`S1`) that already have a corresponding EHA hState `T1` and are connected to an initial pseudo state `S3` by a transition `S2`. For this reason, the type constraint of `S1` prescribes that we expect the instance of a SC `State` in the host graph. Naturally, as `State` is an abstract class thus cannot have instances in the host graph, the `State` pattern node has to be instantiated by the instances of the subclasses of State, i.e. by `SimpleState`s and `CompositetState`s.

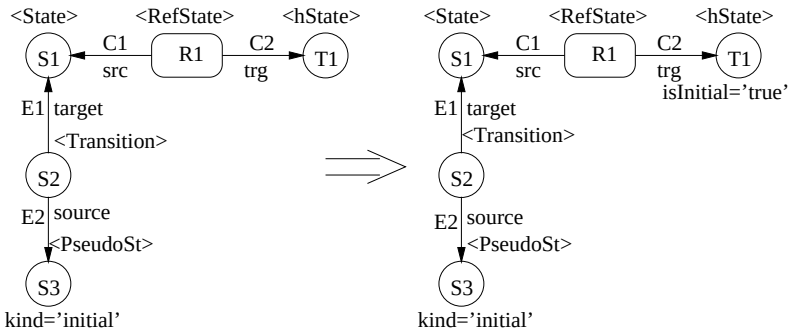


Figure 1.17: The `initStateR` rule

Creating automatons Each composite state `S1` that contains an initial pseudo state `S2` (which implies that `S1` is a non–concurrent composite state) is mapped into a distinct EHA hAutomaton `T1` by applying `automatonR` (Figure 1.18). The composite state and the hAutomaton is connected by a reference node `R1` of type `RefAutomaton` and the corresponding reference edges.

Refining states If a (concurrent) composite state `S1` with a related hState `T1` contains (see the `subvertex` edge) another composite state `S2` (i.e. a region) with a related hAutomaton `T2` then `T2` should be connected to `T1` by a `refined` edge (rule `refinementR1` in Figure 1.19).

Alternatively, if there is a non-concurrent composite state `S1` which is simultaneously related to hState `T1` and a hAutomaton `T2`, `T1` should also be connected to `T2` by a similar `refined` edge when applying `refinementR2` (Figure 1.20).

Please note that the fact of refinement is also indicated by a `refined` edge in the reference metamodel and the requirement of concurrency (or non-concurrency) is implicitly included by the

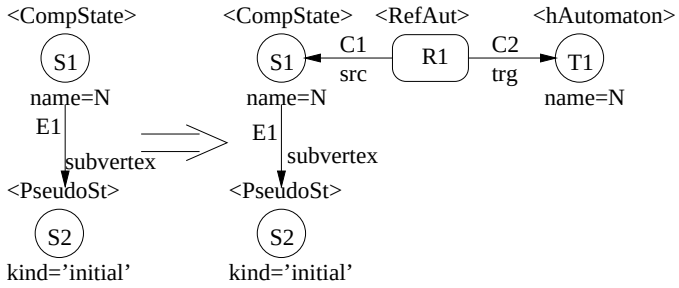


Figure 1.18: The `automatonR` rule

corresponding reference node types of the LHSs (i.e. such conditions were checked during the creation of the reference nodes).

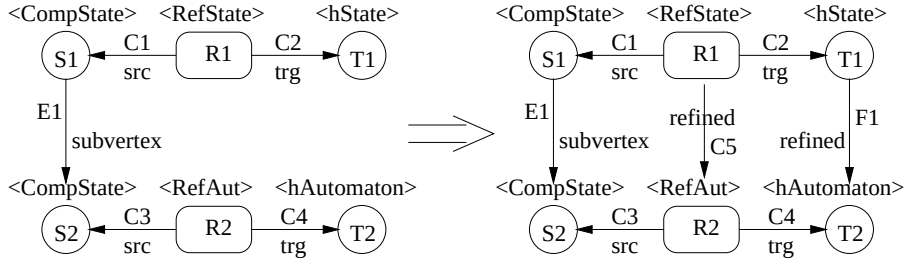


Figure 1.19: The `refinementR1` rule

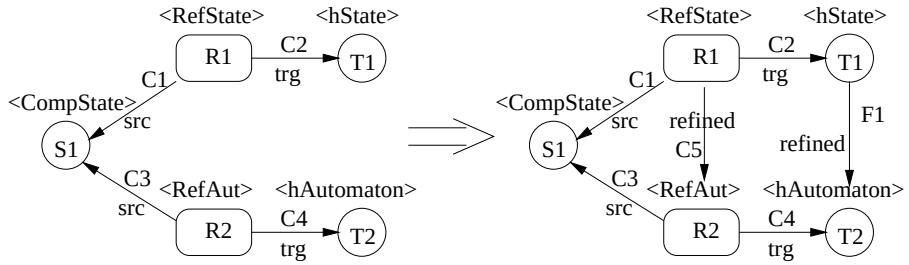


Figure 1.20: The `refinementR2` rule

Connecting states to their automaton Up to now, every hAutomaton is empty, i.e. its hStates are not connected yet by `autState` edges. Thus (according to `autStateR` of Figure 1.21), for each composite state node `S2` refined to a hAutomaton `T2` that has a substate `S1` (either a simple or a composite state) refined to a hState `T1`, this hState `T1` must be linked to its hAutomaton `T1` by an `autState` edge.

Example 1.5.1 After having applied the previous rules in the given order to the statechart of Figure 1.10, the target EHA model should look like the one of Figure 1.22. The interesting parts of the current phase are the following.

- The statechart regions `S4` and `S5` do not have corresponding hStates.

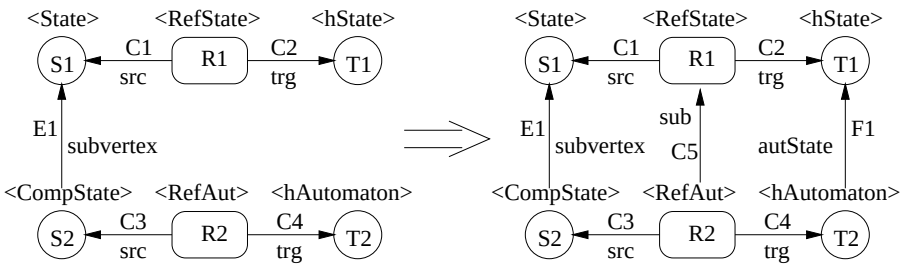


Figure 1.21: The `autStateR` rule

- On the other hand, `S1` does not have a corresponding hAutomaton.

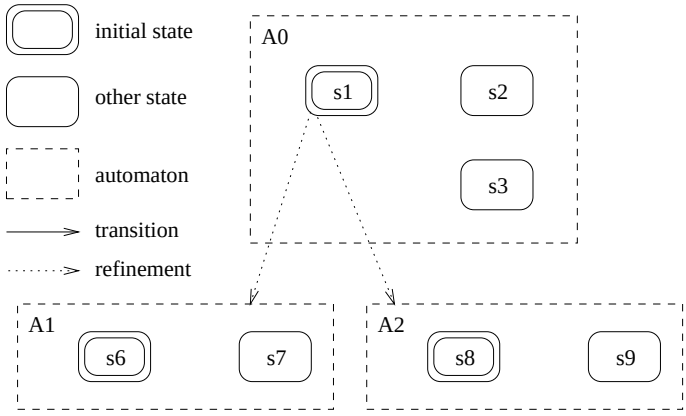


Figure 1.22: The target EHA model after transforming SC states

1.5.2 Transforming Transitions

The next series of rules handles statechart transitions by lifting interlevel transitions to the hAutomaton related to the transition's lowest common ancestor state. Actions, events and guards are also transformed in this phase.

Creating EHA transitions EHA hTransitions are created and linked to their source and target hStates in different phases. As for their creation (rule `transitionR` in Figure 1.23), each real statechart transition `S1` (i.e. that does not lead from an initial pseudo state `S2`) is mapped into a hTransition node `T1` related to each other by a reference node `R1` of type `RefTrans`.

Creating actions, events and guards As the rule for creating and linking actions, events and guards is identical (in its structure), only one of them (creating and linking hActions) is discussed in details. Rules for the others can be obtained by altering the types of nodes and edges according to the metamodels.

According to rule `actionR` (Figure 1.24), for each action `S1` in the statechart, a corresponding hAction `T1` is created for the EHA model (naturally, in addition to a reference node `R1` of type `RefAction`).

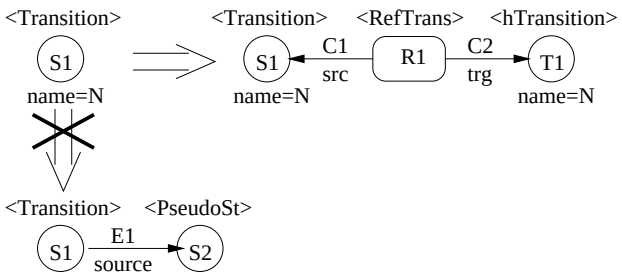


Figure 1.23: The `transitionR` rule

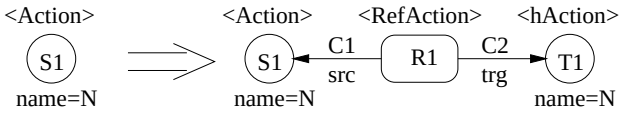


Figure 1.24: The `actionR` rule

Linking actions to transitions For each statechart transition `S1` (derived into an EHA hTransition `T1`) with an action `S2` (with a corresponding hAction `T2`) defining its effect, the hAction node `T2` must be connected to hTransition `T1` by an `ac` edge by the application of rule `actionEffectR` (Figure 1.25).

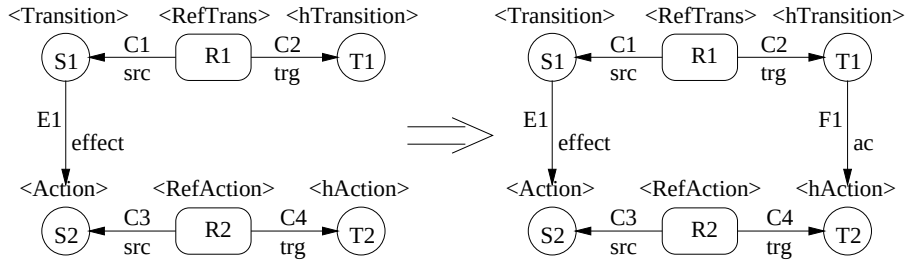


Figure 1.25: The `actionEffectR` rule

Connecting transitions to their automaton The previously created hTransitions are linked to their hAutomaton by applying the `connectAutR` rule (Figure 1.26). Starting from a transition `S1`, its lowest common ancestor state `S2` should be reached via an `lca` edge. After accessing the corresponding hTransition `T1` and hAutomaton `T2` in the EHA model by references, `T1` should be linked to `T2` by a `autTrans` edge.

The interested reader may have noticed that there is no `lca` edge in the metamodel of statecharts. The reason is that the `lca` edge is just an abstraction (i.e. a derived relation between statechart transitions and states), thus not included in the original statechart. Later in this section, these edges will be generated by a corresponding auxiliary transformation rule used for preprocessing the source model before the model transformation.

Linking transitions to states The hTransitions are connected to their main source and target hStates by rule `connectSourceR` (Figure 1.27) and `connectTargetR` (Figure 1.28). For

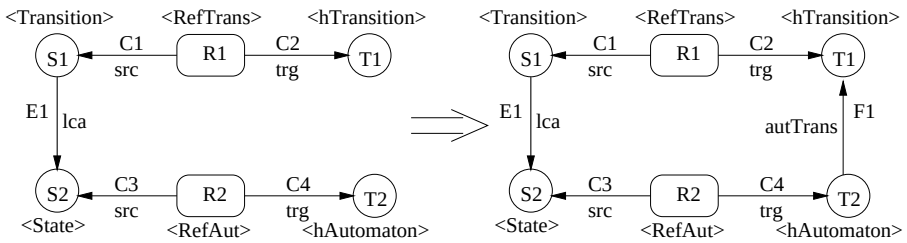


Figure 1.26: The `connectAutR` rule

this reason, additional derived edges (`mainSrc` and `mainTrg`) have been introduced in order to avoid rules of large complexity. For each transition `S1` (with a related hTransition `T1`) its main source (target) state `S2` (with a corresponding hState `T2`) has to be accessed by the auxiliary edge of type `mainSrc` (`mainTrg`). After that, `T2` is linked to `T1` by a `from` (`to`) edge marking the source hState of `T1` in the EHA automaton.

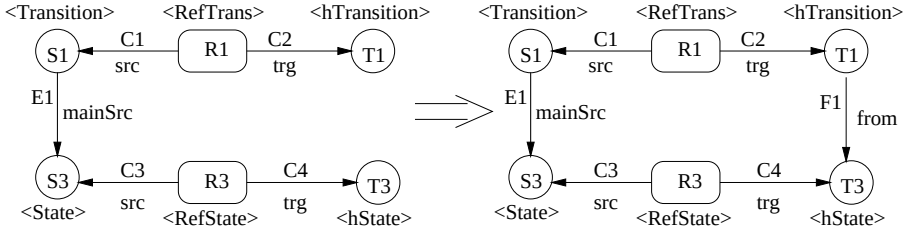


Figure 1.27: The `connectSourceR` rule

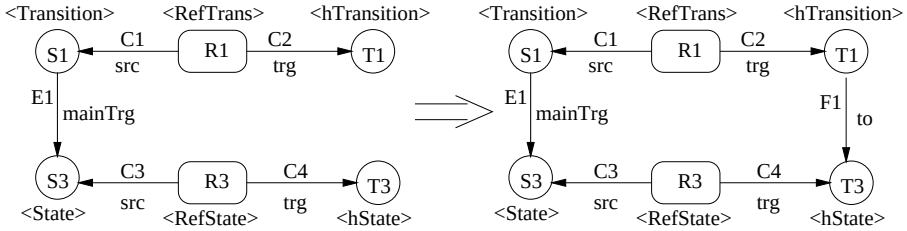


Figure 1.28: The `connectTargetR` rule

At the current point, the transformation has been completed and the target EHA model should look like the one (Figure 1.12) that was expected.

1.5.3 Auxiliary rules

In the previous section, several auxiliary edges (`lca`, `mainSrc` and `mainTrg`) were introduced in order to keep the size of rules manageable. The current section is responsible for preprocessing the UML statechart model, i.e. generating such edges by corresponding transformation rules and units.

Lowest common ancestor, main source and target... All these relations are derived by the application of the `lcaR` rule (depicted in Figure 1.29). However, for expressing these relations, a

further derived edge anc is required leading from each state node to all its ancestors. Please note that this ancestor relation between states include an edge linking a state to itself, i.e. every state is an ancestor of itself.

When deriving the lca edge, those states S4 and S5 have to be found first for each transition S1 with a source state S2 and target state S3 which are ancestors of S2 and S3, respectively, but not matched into identical nodes in the host graph (S4 is not equal to S5). Moreover, S4 and S5 are direct substates of a non-concurrent composite state S6.

After such conditions, S6 is the lowest common ancestor of transition S1 as it is a *common ancestor* (note the subvertex and anc edges) and the isomorphic match condition (i.e. S4 must not share the same node with S5 in the host graph) guarantees that S6 is the *lowest* of such common ancestors. In addition, S4 and S5 are the main source and target states of S1 by definition.

An assertion for this rule is the semantic constraint that transitions are not allowed between different regions of the same concurrent state.

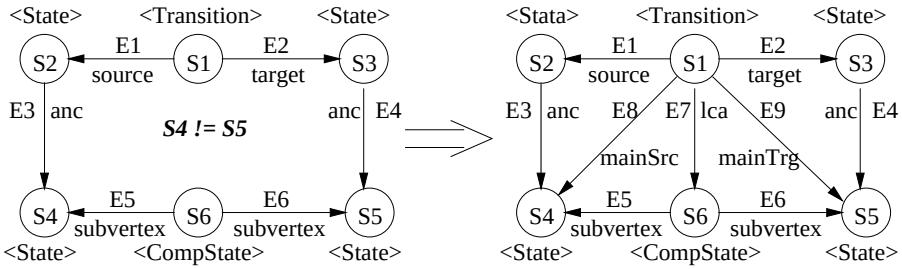


Figure 1.29: The lcaR rule

The rule also handles the case when the main source (and/or main target states) of a transition are equal to its source (target) states since, due to the definition of the ancestor relation (a state is ancestor of itself), nodes S2 and S4 (S3 and S5) may share the same nodes in the host graph.

The ancestor relation (global) For demonstrating the power of programming with transformation units, the ancestor relation will be derived in two different way. The first solution captures the problem globally, i.e. the state hierarchy is traversed in an arbitrary order. The second solution is rather local in the sense that it traverses the state hierarchy in a top–down way by passing the next node to be processed as an attribute to a recursive transformation unit.

The first solution (ancestorTU1 in Figure 1.30) starts with creating an anc edge in parallel with each subvertex edge (ancChildR). Afterwards, the transitive closure of the ancestor relation is calculated by applying ancClosureR as long as possible (the negative condition ensures that each anc edge is generated at most once).

Please note that the using here the *forall* semantics is insufficient as *forall* applies the rule parallelly for each (current) occurrence of the LHS. While, in this case, edges generated by a previous application of the ancClosure rule should also have participate in the next application of the rule, which is a completely different flow of control.

Finally, as each state is an ancestor of itself, an anc edge is added to each state S1 by applying ancSelfR parallelly for all matches.

The ancestor relation (Local) The other solution (ancestorTU2 in Figure 1.31) attaches parameters to several transformation rules and units (input nodes passed as parameters to LHS and

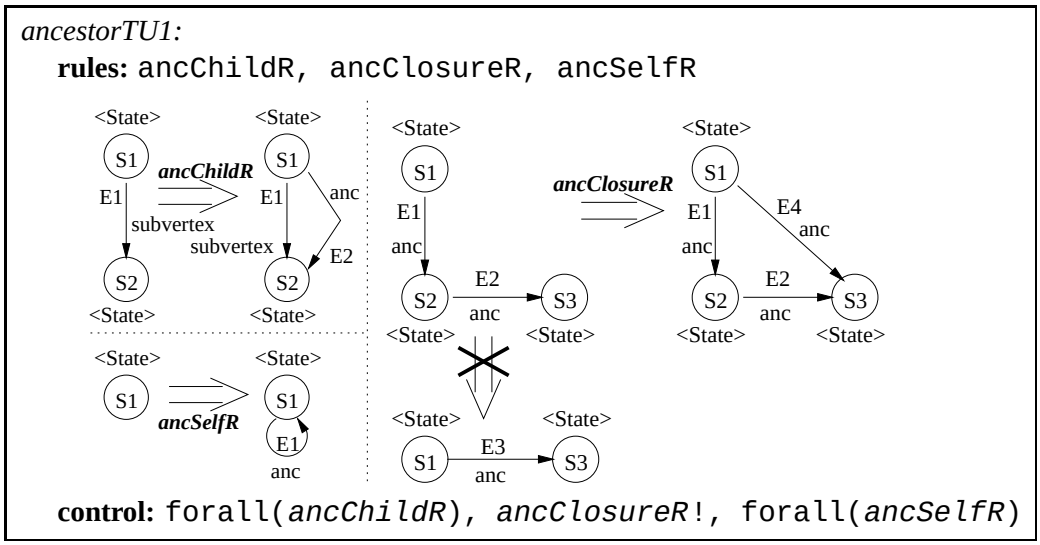


Figure 1.30: Deriving the ancestor relation (ancestorTU1)

output nodes returned from the RHS are depicted grey).

The transformation unit is composed of the rule ancFirstR and calls a further transformation unit ancTU. The state hierarchy is traversed starting from the top state of the statemachine. This top state is selected by applying ancFirst at most once, which also adds the self ancestor edge as side effect. After that, transformation unit ancTU is called with the top state Top as its input attribute. The necessity of the *forall* semantics will be explained together with ancTU.

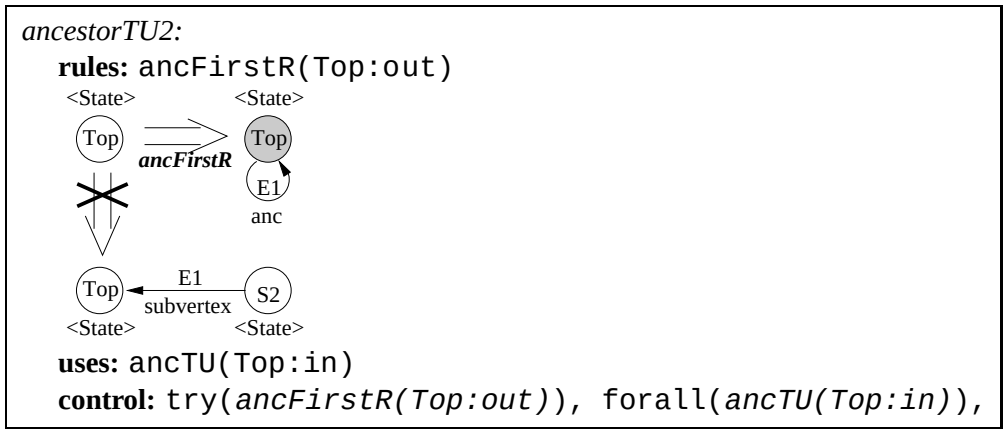


Figure 1.31: Locally deriving the ancestor relation (ancestorTU2)

The transformation unit in Figure 1.32 demonstrates how recursion can be expressed by calling units. According to its control condition, it calls first the ancNextR rule with the input parameter Curr, which is a non–deterministic application of the rule for selecting one direct substate New of the current state. But since ancTU is applied according to the *forall* semantics (see the control condition of ancestorTU2), all the possible matches of ancNextR will be executed parallelly. In other words, the control flow forks to as many branches as many times the ancNextR can be applied.

For each successfully match of node New (of ancNextR), the parallel application of anc-

`Closure2` generates an `anc` edge from all the ancestors of the `Curr` state. Finally, the transformation unit `ancTU` is applied recursively with the substate `New`. If the call of `ancNextR` is not succeeded (i.e. a simple state has been reached in the statemachine), the recursion terminates.

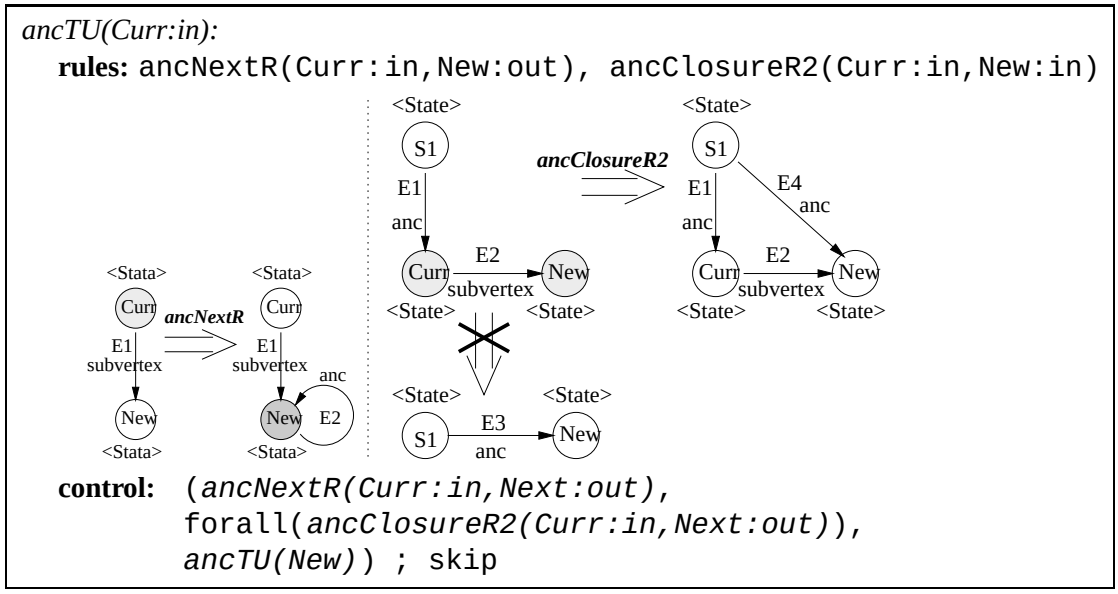


Figure 1.32: Recursion in transformation units (ancTU)

As a result, the state hierarchy of the statemachine is traversed in a top–down way, when a state at level i has been reached, all its ancestors (on level $i - k$) have already been processed.

1.5.4 Control flow of the transformation

In order to obtain a complete specification of the transformation, the control flow will be defined formally in the sequel by means of transformation units.

The entire SC–EHA transformation (Figure 1.33) is carried out by three transformation units: `auxiliaryTU`, `statesTU` and `transitionsTU`, applied in this specific order.

sc2ehaTU:
uses: `auxiliaryTU, statesTU, transitionsTU`
control: `auxiliaryTU, statesTU, transitionsTU`

Figure 1.33: The SC–EHA model transformation

auxiliaryTU:
rules: `lcaR`
uses: `ancestorTU`
control: `ancestorTU, forall(lcaR)`

Figure 1.34: The transformation unit `auxiliaryTU`

statesTU:
rules: `simpleStateR, compositeStateR`
`initStateR, automatonR,`
`refinementR1, refinementR2,`
`autStateR`
control: `forall(simpleStateR), forall(compositeStateR),`
`forall(initStateR), forall(automatonR),`
`forall(refinementR1), forall(refinementR2),`
`forall(autStateR)`

Figure 1.35: The transformation unit `statesTU`

transitionsTU:
rules: `transitionR, actionR,`
`actionEffectR, connectAutR,`
`connectSourceR, connectTargetR`
control: `forall(transitionR), forall(actionR),`
`forall(actionEffectR), forall(connectAutR),`
`forall(connectSourceR), forall(connectTargetR)`

Figure 1.36: The transformation unit transitionsTU

- The unit `auxiliaryTU` (Figure 1.34) is responsible for generating all the `anc`, `lca`, `mainSrc` and `mainTrg` edges.
- The unit `statesTU` (Figure 1.35) – as discussed previously –, applies the following rules parallelly in the definite order: `simpleStateR`, `compositeStateR`, `initStateR`, `automatonR`, `refinementR1`, `refinementR2` and `autStateR`.
- Finally, transformation unit `transitionTU` (Figure 1.36) is responsible for creating transitions and connecting them to their automaton, source and target states, etc. Moreover, actions, events and guards are also handled by them.

1.6 Automatic Program Generation for Model Transformation

Our benchmark transformation may have demonstrated that the creation of model transformation rules, in other words, programming by graph transformation means getting acquainted with a novel, very high level programming paradigm. The current section is concerned with decreasing the level of abstraction by automatically generating a Prolog implementation from the high level specification of model transformation rules and units. In this respect, time and workload can be related to the *design* of transformation rules and not to their *implementation*, thus, the quality of transformations will highly be increased. For the current chapter, the reader's basic knowledge of Prolog is required.

Our automatic program generation approach consist of two major parts.

- As the specification of model transformation is expressed at a very high level, **transformation specific parts** of the automatically generated program must bridge a huge abstraction

gap. Bridging such a gap is always at a high risk for generating erroneous code, moreover, the generated code might implement similar structures more than once thus introducing undesired redundancy to the system.

- To increase the abstraction of programs (and decrease in turn their redundancy), the basic **common control structures** are required to be identified and collected into a separate module. In this respect, the automatically generated program is just a skeleton, which calls the pre-made common routines by proper parameters. Thus, only a rather high-level code is needed to be generated, while the single instructions are performed by the common module.

Let us examine the properties of transformation rules and control structures from such a point of view whether transformation specific or common structures should be dominating in an automated implementation.

- For generating the Prolog code for a model transformation rule, the transformation specific parts will be dominating, as the sequence in which the objects of the LHS have to be matched (which is the most crucial step regarding efficiency) is highly dependent on the rule itself. Moreover, one cannot tell in advance which modifications are prescribed by the rule.
- In contrast to rules, major control flow structures are included in the common module, as the underlying algorithmic skeletons are similar. In this respect, common Prolog predicates will be equipped with rules and transformation units as parameters.

The following sections are thus concerned with (i) an overview of the Prolog data structures used in transformations; (ii) the automatic program generation for rules using the previous structures; (iii) the module for implementing the algorithmic skeletons of control conditions.

1.6.1 Basic Rule Structures in Prolog

The graph model Model transformations manipulate on reference graphs, which graphs are constructed in correspondence with their MOF metamodels. Now, model graphs are transformed into a Prolog term representation and stored as dynamic clauses in an internal fact database, which can be arbitrarily modified at run-time.

The correspondence between graph nodes and Prolog terms are characterized by the following rules.

- From a model graph node of type `type` with an identifier `id`, the predicate `type(id)` is generated (`type` and `id` are considered to be Prolog atoms)
- From a model graph edge of type `type` with its own `id`, source `src` and target `trg` identifiers, the predicate `type(id,src,trg)` is generated.
- From a model graph attribute (attached to the node identified by `nid`) with a name `name`, and having value `value`, the predicate `name(ownid,nid,value)` is generated, where `ownid` is a unique identifier for the attribute (generated automatically).

Each model (either source, target or reference) is stored in a distinct Prolog module. In this way, they can easily be accessed, yet, they are kept separated from each other. Having considered Prolog modules, a term of a model can be accessed by prefixing the term with the identifier of the model.

Example 1.6.1 For instance, we are able to access `simpleState(s1)` of model `sc` (stored in a module with identical name) by `sc:simpleState(s1)`.

Matching graph patterns Graph pattern matching is implemented by using the powerful unification mechanism of Prolog. In rule graphs, the identifiers of nodes and edges are normally variables, which variables get instantiated during the pattern matching process. If a variable is instantiated, it serves as a precondition for all the terms to be accessed later, which contain this specific variable.

Example 1.6.2 Let us consider the LHS of rule `automatonR` (Figure 1.18) prescribing the presence of a composite state containing an initial pseudo state. In the Prolog representation, such a query would look like the following (all the identifiers with capital initials are variables, thus instantiated when applying the rule).

```
sc:compositeState(S1),     % selecting a composite state
sc:subvertex(E1,S1,S2),    % selecting a substate S2 of S1
sc:pseudoState(S2),        % testing the type of S2
sc:kind(A1,S2,'initial'),  % accessing its attribute
```

All the terms are matching a corresponding node or edge type, while the Prolog variables and graph object identifiers are similarly denoted. Although, such a representation of LHS queries might be sufficient at first sight, unfortunately, they raise several problems concerning efficiency and the handling of abstract nodes.

- When the unification of a Prolog term *may* multiply succeed, a choice point is generated, and all the matching terms can be enumerated by backtracking. However, when the execution of the previous program skeleton arrives at to unify `pseudoState`, its variable `S2` is already instantiated, thus (as graph object identifiers are considered to be unique) this call cannot succeed more than once. As a result, an unnecessary choice point is generated, which decreases the efficiency of the program when backtracking is required.
- Similarly, the match of attribute terms can never succeed more than once.
- Graph patterns in the LHS of a rule may contain abstract nodes, i.e. nodes with a corresponding abstract metamodel class. A pure syntactical transformation from LHS patterns to Prolog terms would be unable to handle such matches.

To avoid the previous problems, all the terms in Prolog programs (just in programs not in the model modules) are embedded as a parameter into another predicate, which is responsible for properly handling nodes and edges.

Example 1.6.3 The needed modifications (i.e. the indication of nodes, edges and attributes) for the `automatonR` example are the following:

```
node(sc:compositeState(S1)),     % choice point generated
edge(sc:subvertex(E1,S1,S2)),    % choice point generated
node1(sc:pseudoState(S2)),       % no choice point generated
attr(sc:kind(A1,S2,'initial')),  % no choice point generated
```

Please note the different predicates (`node` and `node1`) for accessing nodes, from which `node1` allows at most one successful match by applying the cut symbol after successfully calling for the pseudo state `S2`. Naturally, a similar distinction can be used for edges but it is unnecessary for attributes (attributes are always cut after being matched). The `node` (and `node1`) predicate is also responsible for accessing the “instances” of abstract classes.

Modifying the models As all the graph objects are stored as Prolog terms, in principle, we need not distinguish between e.g. the addition of a node or an edge. However, to improve the legibility (and consistency) of the automatically generated code, additional predicates were introduced for additions and deletions.

Example 1.6.4 Continuing our `automatonR` example, the additions prescribed by the RHS of the rule take the following form. The first node clause (`eha:hAutomaton(T1)`) is added to the `eha` model, while the second node (`ref:refAut(R1)`) and two edges are added to the reference model. The predicate `add` (a common code predicate) is responsible for generating a unique identifier for objects before being added to the database,

```
add(node(eha:hAutomaton(T1))),  % adding a node to the EHA model
add(node(ref:refAut(R1))),      % adding a node to the Ref model
add(edge(ref:src(C1,R1,S1))),   % adding an edge to the ref model
add(edge(ref:trg(C2,R1,T1))).   % adding another edge to the ref model
```

Negative application condition Negative application conditions prohibit the presence of a matching pattern, thus all the possible extensions of the LHS need to be investigated. If the negative pattern is found then the rule itself should fail, otherwise, if there is no occurrence for the negative pattern, the application of the rule should continue with additions and deletions.

Example 1.6.5 Considering now `compositeStateR` rule as an example for negative conditions, the corresponding Prolog code should resemble to the following (`->` is a cut–like operator used basically in if-then-else structures of Prolog). The entire clause fails if and only if all the four calls succeed.

```
( node1(sc:compositeState(S1)),           % pattern matching for Neg
  edge(sc:subvertex(E1,S2,S1)),
  node1(sc:compositeState(S2)),
  attr(sc:isConcurrent(A2,S2,'true')) ->
  fail                                    % fail if succeeded
;
  true                                    % continue otherwise
)
```

Please note that negative objects that can be mapped to a node or an edge in the LHS (e.g. the composite state `S1`) need not be included in the negative condition for a more efficient implementation (they can be handled as “parameters” for the negative part).

1.6.2 Program Generation for Rules

At the current point, hopefully, the reader has already had an insight on how the generated program of rules should look like. However, the process (know-how) of automatic code generation has not been discussed.

Such a program generator receives a high-level description of model transformation rules as the input, and it should generate the corresponding Prolog program from this specification that implements the transformation.

After a deeper insight, one might notice that this problem can be regarded as a model transformation, having the description of graph transformation rules as the source model and the Prolog code as the target model.

To avoid the abstraction gap (low-level code from high-level specification), moreover, to obtain a language independent transformation (allowing the use of further programming languages instead of Prolog in the future), the program generation process for transformation rules was divided into four sub-transformations (summarized in Figure 1.37).

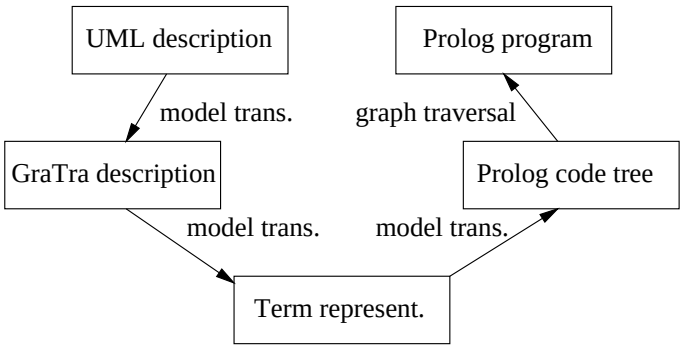


Figure 1.37: Program generation for transformation rules

1. As we stated in the introduction, model transformations are specified in a UML notation (keeping the syntax but overloading its semantics). However, this UML specification is not discussed in the current paper in details.

2. From the UML specification of transformation rules, a syntactical model transformation generates the rules using the terms and the metamodel of graph transformation (GraTra metamodel). This GraTra description (with nodes, edges, attributes etc.) is equivalent with the transformation rules used previously.

3. From this GraTra representation, a semantic transformation generates a logics model containing sequences of terms. This phase also contains an optimization process concerning the ordering of LHS query terms for improving the efficiency of transformations.

4. As the structure of well-formed Prolog programs is typically controlled by BNF expressions (or more generally speaking, by some Chomsky grammars), it does not directly fit into our model transformation approach. But considering that during the parsing of program code, a parse *tree* is generated, we have a graph model at hand for any programming language. Thus, the third model transformation derives a simple tree structure for the Prolog code, containing only terminal and non-terminal graph nodes. Terminal nodes are enriched with the attached text attributes used for storing the pieces of code.

5. Finally, the code generation process simply traverses this code tree and prints the text values stored at each terminal node that is reached. This graph traversal algorithm is general, the same algorithm can be used for different programming languages.

In the following, the program generation process for model transformation rules will be demonstrated on the code generation for a small sample rule (`simpleStateR` of Figure 1.15).

The metamodel of graph transformation Although, model transformation rules are based upon the paradigm of graph transformation, it does not prevent us from regarding it as an ordinary model (like statecharts or automatons), thus, its metamodel (Figure 1.38) can be created by strictly following its definitions (Def. 1.3.11).

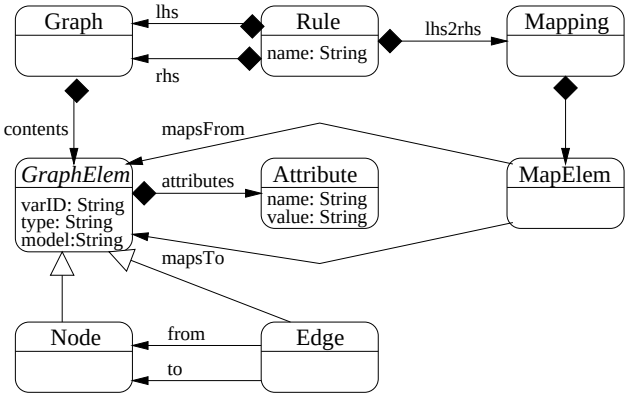


Figure 1.38: The metamodel of graph transformation

- The metamodel (which is a simplified version of the upcoming standard [25]) states that a graph transformation rule is composed of a `lhs` and an `rhs` graph, and a mapping `lhs2rhs` between objects in the LHS and RHS.
- A `Graph` is composed of abstract `GraphElements`, which are either instances of a `Node` or an `Edge`. They possess several attributes, such as `varID` for storing the identifier, `type` for the type of the object and `model` for identifying the model the object belongs to.
- The instances of the class `Attribute` can be attached to `GraphElements`, which class in turn contains two MOF attributes `name` and `value` (please note that the notion of attribute is overloaded).
- A `MapElem`, which is embedded into a `Mapping` contains a reference to an LHS object `mapsFrom`, and a reference to an RHS object. An additional semantic constraint is needed to be set up at the current point prescribing that a map element must not connect LHS nodes to RHS edges and vice versa.

Example 1.6.6 The model graph of the transformation rule `simpleStateR` is depicted in Figure 1.39. Attributes are depicted this time in boxes containing the identifier of their owner node in the top–left corner, while edges of type `contents` have dashed lines. The interested reader may check that this model graph obviously conforms to its metamodel.

Please note that this graph based representation does not explicitly contain any information on the optimal pattern matching of the LHS, thus such an ordering information needs to be added later during the fore-coming model transformation.

The term representation of rules This second model transformation (the first one is considered to be the UML–GraTra transformation which is not discussed in the paper) generates a term representation for graph nodes and edges. Moreover, a nearly optimal ordering of the query terms of the LHS (nearly optimal from the point of view of graph pattern matching) is also provided.

- The metamodel of terms (see Figure 1.40) is composed of `Clauses` on the top of the hierarchy.

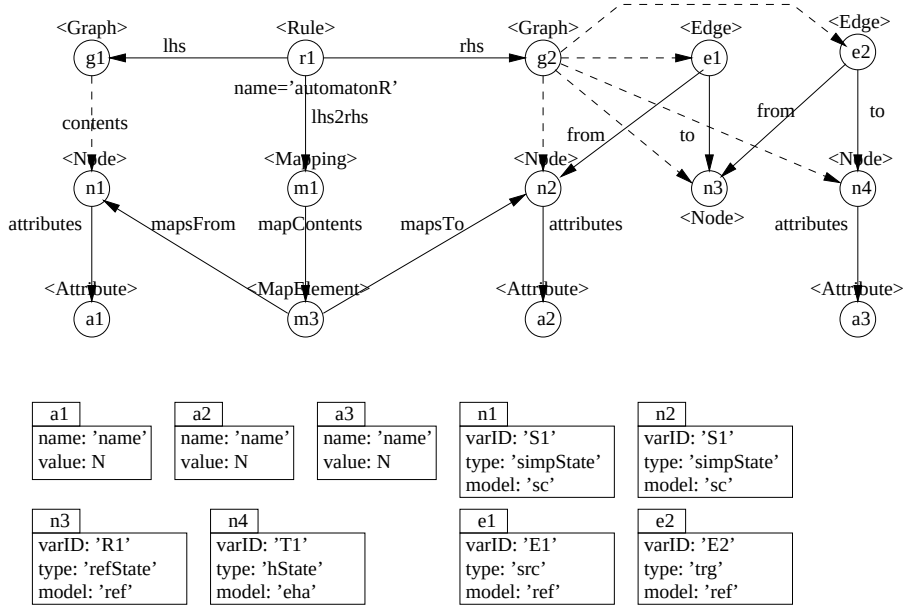


Figure 1.39: The model graph of graph transformation

- The clause is constituted from list of facts (`FactLs`), one for specifying the terms of the LHS query, one for the prescribed additions of terms and one for deletions (in a more detailed metamodel, a fourth `FactLs` would contain terms for the negative condition).
- A `FactLs` object is built up from `Facts` (an abstract class), which is either a `NodeFact`, an `EdgeFact` or an `AttrFact`. The facts are standalone in the sense that e.g. links between nodes end edges (see `from` and `to` relations in the GraTra metamodel) are now encoded into attributes `fromID` and `toID`.
- However, new connections have been introduced for representing the sequence in which these facts are to be generated in the code. The first and the successor terms in the sequence are identified by `first` and `next` edges.

Example 1.6.7 The model graph of the term representation of our running example (the implementation of rule `simpleStateR`) is depicted in Figure 1.41 (attributes are denoted by boxes similarly to the GraTra case).

The objects of the LHS (node `n1` and attribute `a1`) are directly transformed into terms, while a term representation contains only those parts of the RHS graph, which cannot be mapped to a LHS object (such nodes are `n2`, `n3`, `e1`, `e2` in the example). In this way, the term representation is more compact when compared with the corresponding GraTra description.

The ordering of terms is the most semantic part of the transformation as graphs (serving as the input) are structures while the list of terms (the output) is a sequence.

- The sequence of query terms *commences with* the check of objects obtained as *rule parameters* since the images of such nodes and edges are fixed in the host graph.
- If a rule (such as `simpleStateR`) contains no parameters, *an arbitrary node can be matched first*. However, further optimization is possible here to select first that class of

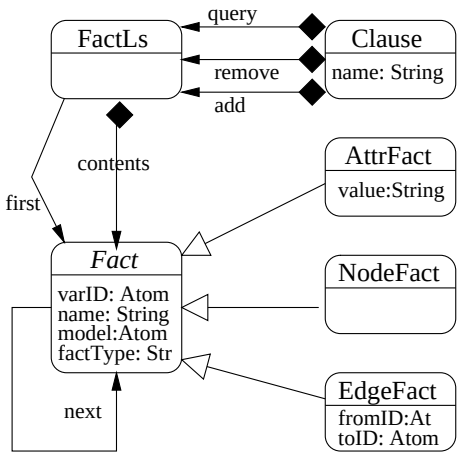


Figure 1.40: The metamodel of terms

nodes which has the least number of instances in order to obtain a search tree having the least number of branches from the root.

- After node has been matched, all of its *attributes are* collected and *checked*.
- Afterwards, the LHS graph is traversed by *matching an edge leading from (or to) a previously matched node* (checking in turn its type), which step selects the target (source) node as well. Then we merely *check* whether *the type of the currently identified* target (source) *node in the host graph* corresponds to its LHS graph specification. Please note that type checking against a metamodel is a less expensive operation than matching graph objects of the LHS.
- Finally, the LHS graph traversal algorithm continues with un-traversed edges. If the LHS graph is composed of more than one components then the algorithm is required to be applied for each component.

The RHS objects are ordered differently in order to maintain the invariant property that the application of a rule always result in a well–formed graph (without dangling edges).

- The deletion of edges should always preceed the removal of nodes. Whenever a node fact is required to be deleted, all the edges connected to that specific node and attributes attached have to be implicitly deleted at the same time. Thus, the remove operation is partially implemented in the common code module.
- In contrast to deletions, the order of adding graph objects is just the opposite. A correct order should start with the addition of nodes followed by the construction of attributes, and finally, the creation of edges. Keeping the specific order ensures that the result is always a graph.

Generating a parse tree From a term representation of graphs, the generation of Prolog facts does not require the bridging of a huge abstraction gap, especially in such a case, when the order of predicates has already been determined. Each syntactic element of Prolog is to be transformed into terminal graph nodes (controlled by the grammar of Prolog).

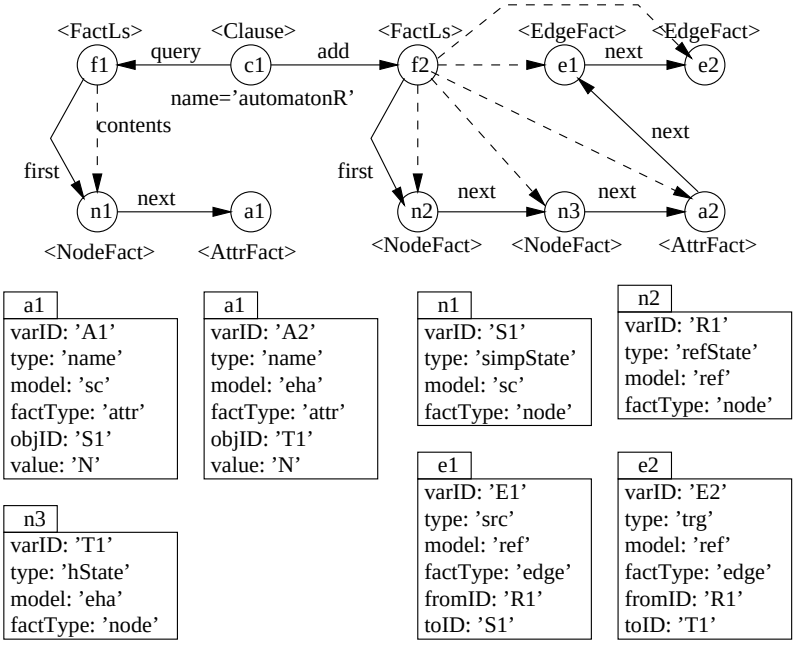


Figure 1.41: The model graph of terms

However, in order to keep our program generation approach language independent, not the final metamodel (in Figure 1.42) is the metamodel of Chomsky grammars, and not the metamodel of Prolog.

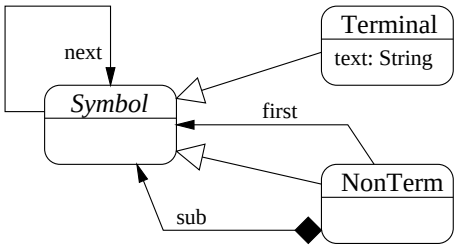


Figure 1.42: The metamodel of parse trees (grammars)

The advantage of such a general solution originates in the fact that only a single graph traversal algorithm is required for the final code generation step for any programming language, while the well–formedness of the parse tree can be verified against traditional context–free grammars. Thus, for this final step, the grammar of the programming language is also needed. A highly simplified grammar of Prolog is printed below.

```
Program ::= Clause  Program    | Clause
Clause  ::= Pred ':-' Terms '.'| Pred.
Pred    ::= atom '(' Arg ')'   | atom
Arg     ::= Pred    | var      | atom
Terms   ::= Pred ',' Terms     | Pred
```

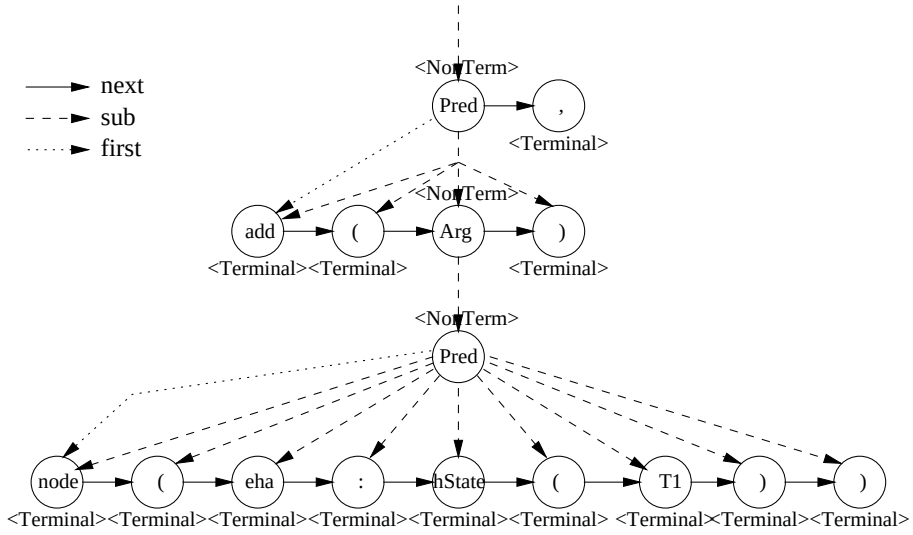


Figure 1.43: A part of the Prolog code graph

Example 1.6.8 The term representation of our rule `simpleStateR` is transformed into a parse tree. Figure 1.43 shows a meaningful part of this tree containing subgraph for the code being generated for the addition of the EHA hState `T1`. (This time the values of `text` attribute of terminal nodes and the names of nonterminals are printed inside the graph node instead of node identifiers.)

The top (depicted) level of the parse tree consists of the `Pred` nonterminals separated by commas as terminals. One level below, the addition is specified by the predicate `add`. This predicate contains another predicate as attribute, which describes the fact (`eha:hState(T1)`) to be added to the database.

The interested reader may check that this tree is a well-formed (part of a) Prolog program by parsing this tree against the grammar and the metamodel.

The generated Prolog code The parse tree is traversed by the following simple algorithm (which visits the tree in a top-down, left-to-right order) in order to generate the final textual representation of rules (i.e. the Prolog code).

1. Start from the root non-terminal of the tree.
2. If a terminal node is reached then print the value of its `text` attribute.
3. If a non-terminal node is reached then
   (a) visit the node identified by the `first` edge (i.e. apply the algorithm recursively from (2))
   (b) while a `next` edge leads from the current node apply the algorithm recursively from (2) to the next node

Example 1.6.9 The algorithm yields the following code when applied to the complete parse tree generated by the previous model transformation step.

```
simpleStateR:-
  % LHS
  node(sc:simpleState(S1)),
  attr(sc:name(A1,S1,N)),
  % RHS
  add(node(ref:refState(R1))),
  add(node(eha:hState(T1))),
  add(attr(eha:name(A2,T1,N))),
  add(edge(ref:src(E1,R1,S1))),
  add(edge(ref:trg(E2,R1,T1))).
```

As a summary, the program generation process of model transformation rules was also designed by model transformations receiving a description of graph transformations as input and yielding the corresponding Prolog code as output. In the following, the implementation of control conditions will be discussed briefly.

1.6.3 Common Program Skeletons

Control structures in transformation units are such parts of the program that have similar underlying skeletons. In the following, control conditions (summarized in Table 1.3) are implemented one by one.

Control condition	Prolog code
skip	`true`
try(Rule)	`try(Rule):-` `  call(Rule), !.`

forall(Rule)	`forall(Rule):-`
	`  call(Rule), fail.`
	`forall(Rule).`

rule1, rule2	`rule1,` `rule2`

rule1; rule2	`( rule1`
	`; rule2`
	`)`

if c then rule1 else rule2	`if_then_else(C,Rule1,Rule2):-`
	`  try(C),`
	`  call(Rule1).`
	`if_then_else(C,Rule1,Rule2):-`
	`  call(Rule2).`

rule1!	`loop(Rule):-`
	`  try(Rule),`
	`  loop(Rule).`
	`loop(Rule).`

rule1 | rule2	effects can be simulated by
	`fork(Rule1,Rule2):-`
	`  forall(Rule1;Rule2).`

Table 1.3: Implementing control conditions

- The **skip** operation is encoded into the always succeeding `true` clause.
- The **at most once** semantics necessitates choice points generated by the successful application of the rule to be cut (the `call` predicate of Prolog is a so–called meta-predicate for being able to call Prolog clauses passed as attributes)
- The **forall** semantics enumerates all the possible matches by causing artificial backtracking (using the always unsuccessful *fail* clause). As the *forall* application of a rule is successful even if the rule cannot be applied at all, a second `forall(Rule)` clause is needed to guarantee that property. This control condition always terminates.
- The **sequence** of two (or more) rules is implemented by the Prolog AND operator (comma). No additional code is required for the common code module.
- For the **non-deterministic choice** of two (or more) rules, the Prolog OR operator (semicolon) is used without additional common code.
- The **if-then-else** structure tries to apply the condition `C`, and if succeeded, calls `Rule1`, and otherwise `Rule2`.
- The clause for the **as long as possible** semantics of a rule tries to apply the rule firts, and calls itself recursively, if the rule were able to be applied. The termination of this control condition cannot always be guaranteed, as it depends on the successful application of the rule.
- The **fork** structure is not implemented yet (as parallelism is not supported in all Prolog systems), however, its effects can be simulated by combining the *forall* and the *non-deterministic choice* operators, as *forall* forces the execution of both non-deterministic branches by backtracking.

For demonstrating how control instructions can be constructed from the previous pieces of code, the encoding control conditions of `ancestorTU2` (the second solution for creating the ancestor relation) are listed below.

Example 1.6.10 In accordance with our expectations, the following piece of code calls `ancFirstR` at most once, then applies `ancTU` for all possible "branches" defined by the non-deterministic choice operator.

```
ancestorTU:-
  try(ancFirstR(Top)),
  forall(ancTU(Top)).

ancTU(Curr):-
  ( ancNextR(Curr,New),
    forall(ancClosureR2(Curr,New)),
    ancTU(New)
  ; true
  ).
```

This piece of code clearly demonstrates that the encoding of control conditions (unlike transformation rules) is rather straightforward, after having introduced the common parts of the program in Table 1.3. The interested reader might also have noticed that the direction of parameters (input or output) is of no importance in Prolog, as the unification mechanism is bi–directional.

1.7 Conclusion

In the current paper, a visual specification method was presented for generally describing and automatically implementing mathematical model transformations in order to integrate UML–based system models and mathematical models of formal verification tools. Due to the large complexity of IT systems, model transformations are supported by an integrated environment, which has a precise theoretical background on the basis of graph transformation but simultaneously follows the main standards of software engineering (such as UML, MOF metamodels and XMI).

Both system and mathematical models were specified by means of MOF metamodels, which provide a semi-formal, visual description on the structure of models, while semantic restrictions (which were not discussed in the current paper) are typically expressed by the Object Constraint Language (OCL).

The manipulation of models is specified by powerful paradigm of graph transformation. For this reason, MOF models are transformed first into a directed, typed and attributed graph representation which combines the source and target models into a common reference graph serving as the input and output for transformations. Model transformation rules themselves are special form of graph transformation rules. Rules can be grouped into transformation units, which provide a structuring mechanism and control conditions for transformation systems with a large number of rules.

The strength of our model transformation approach was demonstrated on an industrial strength example, which generates an extended hierarchical automaton for providing an operational semantics for UML statecharts. Our framework in the current paper provides a formal, high-level specification of this model transformation following the semi-formal guidelines presented first in [16].

Finally, the issues of automatic model generation were addressed by deriving a Prolog program from visual graph transformation rules. This automatic program generation process was also designed via several model transformations, which integrate graph grammars and traditional Chomsky grammars.

Despite the fact that our model transformation approach has proved to be successful for several applications, further future research is needed especially concerning the semantic issues of transformations. Although graph transformation provides a precise means for specifying model transformations of various domains but only guarantees that the result of the transformation is a well-formed graph (*correctness of a single rule application*).

Syntactic correctness of transformations (i.e. the result of the transformation is a well–formed sentence of the target language) were checked by planner algorithms [29]. After being able to precisely describe metamodels (substituting MOF metamodels with a precise metamodelling technique), our attention will turn towards *semantic correctness of transformations* by prescribing special criteria that has to be fulfilled by both source and target models.

Bibliography

[1] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.

[2] A. Bondavalli, M. Dal Cin, D. Latella, and A. Pataricza. High-level Integrated Design Environment for Dependability. WORDS'99, 1999 Workshop on Real-Time Dependable System, 1999.

[3] A. Bondavalli, I. Majzik, and I. Mura. From structural UML diagrams to Timed Petri Nets. *PDCC-CNUCE European ESPRIT Project, HIDE Deliverable 2, Section 4*, November 1998.

[4] A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analyses for supporting design decisions in UML. In *HASE'99: The 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.

[5] E. Canver. Einsatz von Model-Checking zur Analyse von MSCs über Statecharts. Technical report, University of Ulm, April 1999.

[6] M. Dal Cin, G. Huszerl, and K. Kosmidis. Evalution of safety-critical system based on guarded statecharts. In *HASE'99 The 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.

[7] J. Desel, G. Juhas, and R. Lorenz. Process semantics of Petri Nets over partial algebra. In *ICATPN: International Conference on the Application and Theory of Petri Nets 2000*, pages 146–165, 2000.

[8] H. Ehrig, R. Geisler, M. Grosse-Rhode, M. Klar, and S. Mann. On formal semantics and integration of object oriented modeling languages. In *EATCS*, volume 70, pages 77–81. The Formal Specification Column, February 2000.

[9] S. Gyapay and D. Varró. Automatic algorithm generation for visual control structures. Technical report, Budapest University of Technology and Economics, Dept. of Measurement and Information Systems, December 2000.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[11] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[12] M. Jüngel, E. Kindler, and M. Weber. The Petri Net Markup Language. Technical report, Humboldt University, Berlin, August 2000.

[13] H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units — a step into GRACE. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 89–108. Springer, 1996.

[14] S. Kuske. More about control conditions for transformation units. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 323–337, 2000.

[15] J. Larrosa and G. Valiente. Graph pattern matching using constraint satisfaction. In *Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 189–196, April 2000.

[16] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML Statechart Diagrams. In *Proc. IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, February 1999.

[17] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *ASIAN'97 Third Asian Computing Conference. Advances in Computer Science*, volume 1345 of *LNCS*, pages 181–196. Springer-Verlag, 1997.

[18] Object Management Group. *XML Metadata Interchange*, October 1998. http://www.omg.org.

[19] Object Management Group. *Object Constraint Language Specification Version 1.3*, June 1999. http://www.rational.com/uml.

[20] Object Management Group. *UML Semantics Version 1.3*, June 1999. http://www.rational.com/uml.

[21] S. Owre and N. Shankar. The formal semantics of PVS. Technical report, Computer Science Laboratory, SRI International, August 1997.

[22] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1: Foundations. World Scientific, 1997.

[23] A. Schürr. Introduction to PROGRES, an attributed graph grammar based specification language. In M. Nagl, editor, *Graph–Theoretic Concepts in Computer Science*, volume 411 of *LNCS*, pages 151–165, Berlin, 1990. Springer.

[24] A. Schürr. *In [22]*, chapter Programmed Graph Replacement Systems, pages 479–546. World Scientific, 1997.

[25] G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In J. Padberg, editor, *UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, April 2001.

[26] A. Toval Alvárez and J. L. Fernández Alemán. Formally modeling UML and its evolution: A holistic approach. In *Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*. Stanford University, Kluwer Academic Press, September 6-8 2000.

[27] D. Varró. Automatic transformation of UML models. Master's thesis, Budapest University of Technology and Economics, 2000. Available from http://www.inf.mit.bme.hu.

[28] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. In H. Ehrig and G. Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 14–21. Technical University of Berlin, Germany, March 2000.

[29] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, December 2000. Accepted paper.

[30] J. Whittle. Formal approaches to systems analysis using UML. *Journal of Database Management*, 11(4):4–13, 1999.